
CROSSTALK
June 2005 The Journal of Defense Software Engineering Vol. 18 No. 6
RELIABILITY
FUNCTIONALITY
REALITY COMPUTING
DURABILITY

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUN 2005		**00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 18, Number 6, June 2005	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Reality Computing

Best Practices

Software Engineering Technology

Open Forum

ON THE COVER
Cover Design by Kent Bingham.

Departments

CROSSTALK

OC-ALC/ MAS Co-Sponsor	*Kevin Stamey*
OO-ALC/MAS Co-Sponsor	*Randy Hill*
WR-ALC/MAS Co-Sponsor	*Tom Christian*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Kay Jensen*
Phone	(801) 775-5555
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

Oklahoma City-Air Logistics Center (OC-ALC), Ogden-Air Logistics Center (OO-ALC), and Warner Robins-Air Logistics Center (WR-ALC) MAS Software Divisions are the official co-sponsors of CROSSTALK, The Journal of Defense Software Engineering. The MAS Software Divisions and the Software Technology Support Center (STSC) are working jointly to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

The STSC is the publisher of CROSSTALK and provides both editorial oversight and technical review.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us at <crosstalk.staff@hill.af.mil> or use the form on p. 25.

OO-ALC/MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820
(801) 777-5555

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or are endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk> or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Technology Fields Too Slowly

It truly amazes me how technology is woven into the tapestry of our culture. The latest threads of American and global culture are portable computing devices. A year ago I would have never believed that a silly thing called Blackberry was capable of making me do work at 10 p.m. on a regular basis. Ten years ago, none of us knew that it was possible to survive life without a cell phone. Sadly, it also amazes me how slowly the same technology permeates our warfighters' culture, thus capability. The reasons include challenges in securing wireless data, interoperability issues, operating environments, reliability, bureaucracy, and many more. It strikes me as odd that a casual weekend fisherman can see a twig or ditch in 3-D color on depth finders, check the latest weather information with animated Doppler radar, mark his fishing targets within a couple of feet using an onboard relative global positioning system, pull up a navigation map to safely return to the landing, call his wife to coordinate boat docking, and order replacements for all the lures he snagged while waiting for his trailer, yet many of our warfighters don't have the breadth of a fisherman's capability in their hands. As you read this month's CrossTalk and think about portable computing devices, ask yourself, "How can we do our part to effectively field this technology?"

Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor

Computing Permeates Our Society

This month's CrossTalk focuses on information you should know whether you're buying a personal digital assistant, protecting your home computer, designing or writing programs, or improving your processes. It is interesting to watch some technologies fade away while others continue to evolve. It appears only the strong survive, but the required strengths vary from technology to technology. In the debate over video formats, many believed Beta was better, but VHS had a stronger market niche. Today, basic cell phones are passé – everyone wants a picture phone (or so the commercials imply).

We asked Col. Kenneth L. Alford to write his article, *Handheld Computing*, due to the interesting work at the U.S. Military Academy where he taught. Examples include network analysis that looks for new ways to automatically recognize computer network attacks, and a computerized mapping project that shows how digital mapping, terrain, and unit data can be integrated to show warfighters a more complete picture on the ground. Terry Bollinger's article, *How to Secure Windows PCs and Laptops*, was provided to help both home computer users and companies better secure their computers. We also target more advanced development with Rick McCabe's and Mike Polen's article, *Effective Practices for Object-Oriented System Software Architecting.* Lynn Sanderfer, David Herron, David Garmus, Dr. Babak Makkinejad, and Paul Kimmerly also provide basic practices and advice aimed at both novice and advanced programmers. Since only the strong survive in today's fast-moving technologies, which do you want to keep and which do you want to do away with?

Elizabeth Starrett
Associate Publisher

Note of Appreciation: We at CrossTalk extend our appreciation to Walt Lipke as he retires from government service at the end of June. Throughout the years, Mr. Lipke has been a great supporter of CrossTalk via the articles he shared and his key role in helping us secure funds for CrossTalk until our new sponsors started their support. We will miss Mr. Lipke's support in the government sector, but look forward to more words of wisdom via future CrossTalk articles.

Handheld Computing

Col. Kenneth L. Alford, Ph.D.
U.S. Army

Handheld computers, especially personal digital assistants (PDAs), are increasingly being used throughout the Department of Defense. This article highlights some of the ways that PDAs are used today and are envisioned for the future. It outlines some of the considerations involved in a PDA procurement, discusses four tools for developing PDA resource materials – programming tools, hypertext markup language- and eXtensible markup language-based tools, text tools, and calendar tools – outlines some of the benefits and challenges associated with using PDAs, and shares several lessons learned.

Computing power continues to increase in capacity and decrease in cost and size. Today's low-cost handheld computers and personal digital assistants (PDAs) allow computing resources to be almost ubiquitous. This article illustrates how handheld computing is changing the way some Department of Defense (DoD) organizations *do business* and discusses how you may take advantage of the capabilities that are available through this technology.

Example Handheld Programs

PDAs, which are defined in DoD Directive 8100.2 as "a generic term for a class of small, easily carried electronic devices used to store and retrieve information" [2], were at one time viewed as little more than novelties. Yet, with approximately 20 million PDAs shipped last year, business and government groups continue to find new ways to use them.

Here are some of the many ways that PDA technology is currently being used within the DoD:

- The Pocket-Sized Forward Entry Device (PFED) is a ruggedized PDA designed for forward observers, artillery fire direction, and target acquisition missions. With built-in laser range-finding and global positioning system (GPS) hardware, PFED users can issue a call-for-fire request in less than 10 seconds – compared to 45 seconds or more when using other systems [3].
- Security guards at Wright-Patterson Air Force Base use PDAs at the gates and on patrols. Prior to receiving PDAs, guards were provided thick binders containing daily event lists with guest and sponsor contact information. Only three of the 10 gates on base received a binder each day. Visitors who arrived at one of the other seven gates had to wait for someone to confirm if they were allowed on base. Now each guard has a copy of the complete daily list. Using their PDAs, guards can also provide visitors with maps and directions to their destination on base [4].
- The Commanders Digital Assistant is a ruggedized PDA that is being fielded to combat troops; it has already seen field use with the 82nd Airborne in Iraq. It integrates dismounted troops into the *Blue Force Tracking* system, enables commanders to distribute orders, and provides several additional capabilities [5, 6].
- The Coast Guard uses PDAs to help automate the task of inspecting fishing vessels. Based on answers to 16 questions, the PDA generates a customized checklist of safety requirements for firefighting, lifesaving, and bridge equipment appropriate to each vessel. As boarding officers click on each requirement, the PDA displays detailed specifications. Officers then check off whether items are found to be satisfactory or are in violation of federal law. Instead of laboring to determine whether the law requires a particular ship to carry one of three different types of life rafts, an inflatable buoyant apparatus, life float, or nothing at all, boarding officers, with PDAs in hand, now spend their energy on, for example, inspecting life rafts to ensure that they are properly set up to release should a ship sink [7].
- In 2004, the Marine Corps awarded a $2.9 million contract to purchase rugged PDAs (RPDAs) designed to increase tactical awareness. This follows a previous $12 million award that supplied RPDAs to several branches of the U.S. military. These devices offer navigation, tactical digital messaging, remote radio control, GPS, wireless local area network (LAN) and Bluetooth capabilities [8].
- All West Point cadets purchase a PDA. Students and faculty members use PDAs to store schedules, class assignments, reference documents, and other instructional materials. PDAs have also been incorporated into some senior engineering design projects.
- Army medical units in Iraq use PDAs to collect patient information.
- Los Alamos National Laboratory's researchers have developed a PDA radiation detector [4].

> *"… I sat down with a group of Marines in California who had just returned from the Iraq war. At one point in our conversation, I asked the Marines in the room to raise their hands if they had either a cellular telephone or a PDA. Every hand went up."*
>
> *—Vice Adm. Herbert A. Browne, U.S. Navy (Ret.)* [1]

There are many other programs and projects throughout the DoD that could also be listed. Many service members even receive a PDA as part of their unit's standard issue.

Benefits

As the previous examples illustrate, there are many ways to effectively incorporate PDAs into the workplace and many reasons for doing so. PDAs offer a variety of

advantages over other automated solutions. For example, given the right work environment, PDAs can do the following:

- Increase worker productivity.
- Improve customer support.
- Reduce time spent searching for information.
- Allow users to take computing power and large amounts of information to places where they could not previously go.
- Increase accuracy.
- Save lives.

This is because PDAs are versatile, portable, inexpensive, and easy to use.

Retired Vice Adm. Herbert A. Browne recently noted the following:

> Virtually all of the lessons learned from Operation Iraqi Freedom include comments about the speed of the forces' advance exceeding the speed with which line-of-sight communications could keep up. Command and control (C2) of our forces was challenged by this rapid mobility, and wireless technology is viewed as the most likely solution to getting collected information and C2 down to the tactical, *trigger-pulling* level. [1]

PDAs can offer solutions to some of those challenges.

Concerns

Like other forms of automation, introducing PDAs into the work environment comes with a potential downside as well. Here are some of the concerns associated with using PDAs:

- PDAs have limited data input capabilities, especially for typing-intensive work.
- There are additional maintenance and cost requirements.
- Non-ruggedized PDAs can be easily broken.
- PDAs may introduce additional training costs in both time and dollars.
- It is difficult for many users to separate business and personal use of PDAs.
- There are numerous security concerns.

Security

PDAs can introduce multiple security challenges. Due to their small size, for example, PDAs have an increased risk of loss or theft. PDAs that wirelessly connect to government networks create additional obvious security concerns.

On April 14, 2004, Deputy Secretary of Defense Paul Wolfowitz signed Defense Department Directive 8100.2 "Use of Commercial Wireless Devices, Services, and Technologies in the Department of Defense (DoD) Global Information Grid (GIG)" [2]. The directive's guidelines apply to all commercial wireless devices (including PDAs). It establishes "policy and assigns responsibilities for the use of commercial wireless devices, services, and technologies in the DoD GIG."

Hardware Considerations

The use of business and government (*enterprise*) PDAs has been on the rise for the past several years. The Gartner Group's research estimates that more than 4 million enterprise PDAs were in service during 2004, and they estimate that number will rise to 6 million by 2008. At the same time, the total cost of ownership (TCO) per year of a business PDA has dropped 28 percent – from almost $2,700 in 1999 to $1,946 in 2004 [9]. (These figures are based on a two-year amortization rate.)

"The Commanders Digital Assistant is a ruggedized PDA that is being fielded to combat troops; it has already seen field use with the 82nd Airborne in Iraq."

The popularity of PDAs has resulted in a wide variety of devices being offered in the marketplace. Government users considering PDA purchases have several decisions they must make regarding the device they will use. Decisions must be made regarding the processor, operating system, wireless technologies, screen characteristics, battery life, expansion memory, input options, and durability (business versus ruggedized models), among other questions.

Processor

Most PDA processors are made by Intel, Samsung, or Texas Instruments; PDA processor speeds currently range from 16MHz to over 500MHz.

Operating System

Most PDAs run on one of the following operating systems:

1. **Palm OS** (the latest version is Palm OS Cobalt, previously known as Palm OS 6).
2. **Windows Mobile** (which comes in three profiles: Pocket PCs, Pocket PC Phone Edition, and Windows Mobile for Smartphones).
3. **Symbian OS** (the leading operating system for Smartphones).
4. **RIM** (used in Blackberry devices).

Wireless Technologies

An increasing number of PDAs are being sold with one or more of these wireless technologies [10]:

1. **Bluetooth.** This is an industry specification for wireless communications that uses short-range radio technology. Bluetooth version 1.2 throughput is typically in the 400-500 kilobytes per second (Kbps) range. Bluetooth does not have any native support for Internet Protocol (IP) that means it does not currently support tactical computer processor (TCP)/IP or wireless LAN applications very well. It is better suited for connecting PDAs, cell phones, and PCs during short intervals. Bluetooth certification means that an individual product has been tested for compliance with the Bluetooth specification; it does not guarantee that it will be compatible with other Bluetooth-enabled products.
2. **802.11/ Wireless LAN (WLAN).** On an 802.11 WLAN, most PDAs adhere to the 802.11b specification that provides 11 megabytes per second (Mbps) transmission rates and throughput in the 4-6 Mbps range.
3. **Global System for Mobile communications (GSM), General Packet Radio Service (GPRS), and GSM Evolution.** The GSM is a European digital cellular phone standard. GSM 1900, the North American version of GSM, generally provides throughput in the 56 Kbps (or less) range. GPRS is an overlay to GSM networks; the maximum speed is theoretically 171.2 Kbps, but actual user experience is usually limited to 56 Kbps or less. GSM Evolution is a radio interface technology for mobile services; while data throughput rates of approximately 384 Kbps are theoretically possible, users more frequently report throughput closer to 64 Kbps.
4. **Code-Division Multiple Access (CDMA).** This is a spread-spectrum digital cellular technology. It was first used during World War II and became

commercially available in 1995. The latest CDMA standard provides voice and data rates up to 2 Mbps.

Screen Characteristics

The screen size, resolution, backlighting capability, and color should all be considered during any purchase evaluation.

Battery Life

Battery life is an important element in determining which PDA to purchase. Nearly all PDAs today offer rechargeable batteries. The two main battery types are lithium ion and lithium polymer. Most PDAs offer a rechargeable lithium ion battery.

As would be expected, actual battery life depends on a wide variety of factors: screen size, resolution, use of backlighting, data intensity of applications, etc.

Expansion Memory

While PDAs have generally offered a relatively small amount of memory (in the 8-64 megabytes range), new models are offering 256 megabytes with forecasts and expectations for more. Data-intensive PDA applications often require additional memory, and most PDAs offer at least one external expansion slot for additional memory. Several memory card expansion options are available [9]:

1. **Secure Digital (SD)** cards are small and are used in PDAs, digital video camcorders, digital cameras, audio players and mobile phones. Cards are currently available to store up to 1 gigabyte.
2. **Secure Digital Input/Output (SDIO)** provides improved multimedia and device management. SDIO cards generally support PDA peripherals (such as GPS receivers, Wi-Fi, digital cameras, and Bluetooth).
3. **Multimedia Memory Cards (MMC)** expansion slots are supported by many PDAs and are available in storage sizes up to 1 gigabyte.
4. **Compact Flash (CF),** at one time, was the most popular PDA expansion card format. Some PDAs allow users to add up to 6 gigabytes of expansion memory.
5. **Memory Stick.** This is a proprietary format (from Sony). Memory sticks currently have a storage capacity up to 2 gigabytes.

Input Options

In addition to standard PDA touch-screen data entry (either through point-and-touch or PDA graffiti recognition software), many PDAs offer small built-in or attachable QWERTY (standard) keyboards. At least one company has developed an exciting prototype technology:

> … that lets users of PDAs and similar mobile devices put data into their handheld systems simply by typing on an image of a standard-size keyboard projected onto a desktop or other surface. The *electronic perception* technology captures the user's finger motions via emitted light photons that form 3-D real-time images that are then processed and translated into keystrokes. [12]

Durability

Ruggedized PDAs (also known as industrial handhelds) must meet additional specifications. For example, they must be able to survive significant drops (from one meter or greater) onto a concrete floor, water tests, heat and cold tests, and tolerance for electromagnetic shock. Ruggedized handhelds generally cost between 50 percent and 200 percent more than business PDAs.

> *"Every decade has some word associated with it. In the '80s, it was the PC. In the '90s, it was the Internet. For the rest of this decade, the key word is going to be convergence."*
>
> *—Steve Case, AOL Time Warner, Chairman of the Board* [11]

The Gartner Group estimates that the TCO in 2004 of a ruggedized PDA was $2,002 per year (almost the same as the $1,946 TCO for a business PDA) [9]. These figures are based on a three-year product life, but many ruggedized users keep their PDAs longer than three years, which lowers the ruggedized TCO cost. Because they are designed to withstand rougher treatment, ruggedized PDAs tend to have a longer useful life than business PDAs.

Convergence

Digital convergence – defined here as the *coming together* of two or more technologies – is rapidly occurring in the field of handheld computing. In many cases, PDAs and cellular phones have merged and are sold in a single device known as a Smartphone. In the third quarter of 2004, Smartphone shipments exceeded those of PDAs (3.96 million to 2.86 million units) [13].

New devices, new capabilities, and new technology will continue to appear. The DoD should embrace and incorporate, whenever possible, the new opportunities that these advances will provide.

Developing PDA Tools

PDAs are increasingly being used throughout the DoD, and this is especially true at many educational and training locations. For example, at the U.S. Military Academy, PDAs are becoming an effective instrument in an instructor's toolkit.

There are several no-cost, low-cost, and commercial products available to increase the functionality and usability of personal digital assistants in work and training settings. There are four basic ways that PDA resource materials can be developed: (1) programming tools, (2) hypertext markup language (HTML)- and eXtensible markup language (XML)-based tools, (3) text tools, and (4) calendar tools.

Programming Tools

Several software development environments exist for creating applications for personal digital assistants. Introductory information can be found online at numerous Web sites[1]. Java – whether in the form of Kilobyte Virtual Machine (KVM), Kawt (an implementation of the Abstract Window Toolkit for the KVM), J9 (IBM's virtual machine that is supported by Visual Age Micro Edition), or any number of other flavors – is currently one of the most popular languages for programming handheld applications. C, C++, versions of Basic and Visual Basic, Compact Application Solution Language (CASL), Pascal, Forth, Scheme, Smalltalk, and several other proprietary scripting tools are also available [14].

Application development is appropriate when existing PDA software resources cannot adequately satisfy existing requirements. At West Point, for example, plebes (freshmen) must be able to recite the number of days that remain until each football game, spring leave, graduation, and other notable cadet activities. Faculty in the Department of Electrical Engineering and Computer

Science created a small PDA program called *The Days* that automates and simplifies this daily ritual for freshmen students.

HTML- and XML-Based Tools

HTML and XML files can be easily formatted to be viewed on a PDA. There are numerous ways this capability can be used to support teaching and other military training. For example, instructors at West Point have formatted all of the following into HTML and/or XML pages to be read on PDAs: course syllabus and guidelines, course schedules, reading assignments, project and other course assignment files, supplemental reading assignments, class handouts, review information for exams, self-administered non-graded quizzes, Web pages, Web sites, other HTML documents, and student projects. In other work environments, it may be appropriate to place copies of regulations, field manuals, directives, maps, and directions on PDAs.

Two of the most popular programs for transforming Web-ready pages into PDA documents are Plucker[2], a freeware program, and AvantGo[3], a commercial product. Plucker software, for example, allows users to easily create and update PDA-readable versions of a single Web page, multiple Web pages, or entire Web sites. Users can configure graphic quality and size, timing of file updates, depth of search, and numerous other options.

Text Tools

As an alternative to formatting text into an HTML or XML formatted document, there are several text-based tools, each with its own unique file format that can be used to develop PDA-readable documents.

There are several no-cost and low-cost alternatives available for developing PDA documents. One of the quickest and easiest ways to send text and small documents to many PDAs is to use the *Memo Pad* feature found in Microsoft Outlook and several PDA synchronization programs. Using Memo Pad is a very basic solution, but it does have the advantage that students can modify original text they receive. Memo notes are useful when students are asked to modify and return a document or assignment.

To obtain increased document functionality, instructors can turn to freeware or commercial software products, such as PalmReader[4] or Adobe Acrobat[5]. PalmReader provides all of the software and instructions necessary to create finished PDA-readable documents with text and limited graphics. Copies of the reader software are available for the Windows, Macintosh, Palm OS, and Windows Mobile operating systems. An advantage of using PalmReader or Adobe Acrobat is that the same document can be read on platforms with all four supported operating systems. PalmReader uses a unique tag-based description language that supports text styles, links, graphics, sidebars, footnotes, bookmarking, notes, and other features.

PalmReader, Adobe Acrobat, and a variety of similar programs can be used to create functional and extremely useful documents and electronic books for handheld computers.

Calendar Tools

The ability of handheld computers to hot synch with personal information management software, calendar software, and other information located on personal computers makes calendar and schedule applications a natural target for coursework development.

> *"It is important to realize that the next generation of federal workers will not remember a time when computers and connected environments did not exist."*

Faculty and students at the U.S. Military Academy use Course Hour Appointment for Outlook Scheduler (CHAOS) – a Microsoft Visual Basic program for entering lesson dates, titles, and assignment information into their personal and handheld computers. CHAOS-generated files are loaded into the Microsoft Outlook calendar and then synchronized with the calendar program on student and faculty PDAs. Faculty and students at the U.S. Military Academy have used this method to create and distribute calendars and schedules for student clubs, student project teams, and a variety of other groups.

Lessons Learned

Instructors at the U.S. Military Academy have been creating PDA instructional content for several years. Here are some of the lessons they have learned in developing applications and resources for PDAs:

- Be aware of size constraints. PDA storage capacity is rapidly improving, but it is still a constrained development environment.
- Use graphics sparingly and ensure that they have been optimized.
- Double-check everything, especially links and graphics, *before* you distribute PDA programs and products.
- Whenever possible, provide non-handheld alternatives for resources you develop for PDAs.
- Recognize different PDA operating systems, screen resolutions, and color capability. Reference to specific colors in images, for example, is meaningless for users who have a 16-grayscale PDA screen.
- Eliminate screen scrolling on PDAs whenever possible; it is tedious and time-consuming.
- Limit the number of PDA file-reading programs you require users to load on their PDA. PalmReader, Plucker, and Adobe Acrobat, for example, are all excellent programs, but requiring users to load all of them in order to read a wide variety of materials could require an unreasonable amount of valuable PDA memory space.
- Concentrate first and foremost on usability. The best content available can be rendered almost useless by a poor presentation.

Future Handheld Programs

Numerous DoD handheld computing applications are currently being developed. Here is a glimpse of some of the applications that we should see in the future:

- The Protect America system is being designed to give military commanders and force protection personnel from U.S. Customs, the Secret Service, the U.S. Coast Guard, the Transportation Security Administration, the Federal Bureau of Investigation, and others a common information-sharing tool [15].
- The Seismic Landmine Detection System would send seismic waves through a minefield, slightly moving the earth and items buried beneath. A radar sensor, connected to a PDA, will measure ground displacement to locate plastic anti-personnel or anti-tank mines [16].
- Technologies are being developed that will enable computers and PDAs

"to accept gestures, motions, speech, and facial expressions as data input methods … designed to facilitate silent troop communication during combat" [11].

- PDAs are being tested to serve nuclear, biological, and chemical detection roles in future conflicts.

Many other exciting applications will be delivered, as well.

Conclusion

A great deal is being written about the large personnel turnover that the federal government will experience during the next decade as the existing workforce retires in substantial numbers. It is important to realize that the next generation of federal workers will not remember a time when computers and connected environments did not exist. A recent report from the Gartner Group estimates (with an 80 percent probability rating) the following by 2007:

> Seventy percent of all levels of Western governments will double the turnover of new employees due to worker dissatisfaction with the technology infrastructure. … The generation that is about to enter the government workforce thinks differently. They use cellular phones and instant messaging as a matter of course. Facts are available in seconds from online resources. Projects are done in teams. High school and college campuses have wireless access in common areas. Dormitory rooms have broadband access. In the meantime, governments treat IT [information technology] as overhead, and consider Internet access too easy to abuse and instant messaging a time waster. However, the incoming workforce expects to be able to work *anywhere*, *anytime*. It is accustomed to receiving information in seconds, not in quarterly reports. It will not work in an environment that cannot support these requirements. Economic situations or a sense of responsibility may draw workers to government jobs, but unless the technology environment changes the next generation of workers will not perform well and will be highly dissatisfied. As a result, government agencies will lose access to this pool of talent. [17]

Defense personnel should be able to use the best tools available to complete their mission. In many instances, the best tool available will be a handheld computing device such as a PDA. Security concerns regarding the use of PDAs can and must be resolved.◆

References

1. Browne, Herbert A. "Wireless Technologies Are the New Information Revolution." Signal Dec. 2003 <www.afcea.org/signal/articles/anmviewer.asp?a=46>.
2. Department of Defense. "DoD Directive 8100.2: Use of Commercial Wireless Devices, Services, and Technologies in the Department of Defense (DoD) Global Information Grid (GIG)." Washington: DoD, 14 Apr. 2004. <www.dtic.mil/whs/directives/corres/html/81002.htm>.
3. "Pocket Sized Forward Entry Device (PFED)." Defense Update 2005: Issue 1 <www.defense-update.com/products/p/pfed.htm>.
4. Lilie, Cheryl. "Personal Assistants Aid Security." Signal. Feb. 2004. <www.afcea.org/signal/articles/anmviewer.asp?a=10>.
5. "Land Warrior System." Defense Update. <www.defense-update.com/features/du-4-04/land-warrior-2.htm>.
6. "Survival of the Fittest," Jane's Defense Weekly 21 May 2003: 25-28.
7. Rossett, Allison, and Erica Mohr. "Performance Support Tools. The U.S. Coast Guard Uses PSTs …" Business and Management Practices, 58.2 (Feb. 2004): 34.
8. "Marines to Receive Rugged PDAs." Defense Daily International 4.9 (12 Mar. 2004):1.
9. Redmon, P. "Total Cost of Ownership Is Down for Business PDAs." Gartner Research Note COM-22-1757 21 Apr. 2004.
10. Troni, Federica, and Roberta Cozza. "Personal Digital Assistants: Overview." Gartner Technology Overview DPRO-90774. 29 Mar. 2004.
11. Blosch, Marcus. "Convergence: Fusion or Fantasy?" Gartner Report Jan. 2002.
12. Vijayan, Jaikumar. "User Interfaces." Computerworld 38.32 (Aug. 2004): 28.
13. Gartner Group. "3Q04 Smartphone Shipments Overtake PDAs." Gartner Dataquest Alert 22 Dec. 2004.
14. Jones, N. "Choosing a Pocket Programming Language," Gartner Research Note DF-16-1282 11 June 2002.
15. Kellogg Jr., Joseph K., and Mark Powell. "Protecting America With Information Technology." Signal June 2003. <www.afcea.org/signal/articles/anmviewer.asp?a=207>.
16. Lilie, Cheryl. "Land Mine Detector Makes Waves." Signal July 2004. <www.afcea.org/signal/articles/anmviewer.asp?a=222>.
17. Baum, C., et al. "Predicts 2005: Government Ramps Up IT." Gartner Commentary 1 Nov. 2004.

Notes

1. See <www.onjava.com/pub/-a/onjava/2001/03/15/java_palm.html>, for example.
2. The Plucker software home page (which contains programs, instructions, samples, and source code) is <www.plkr.org/index.plkr>.
3. AvantGo, a commercial software product that is often bundled with other PDA-provided software, is available at <www.avantgo.com>.
4. PalmReader software is available at <www.palmdigitalmedia.com/-product/reader/browse/free>.
5. Adobe Acrobat software for the PDA is available at <www.adobe.com>.

About the Author

Col. Kenneth L. Alford, Ph.D., is a professor at the Industrial College of the Armed Forces at the National Defense University in Washington, D.C. He has served 26 years in the U.S. Army as a personnel, automation, and acquisition officer in a wide variety of duty assignments, including his previous position as an associate professor in the Department of Electrical Engineering and Computer Science at the United States Military Academy, West Point, N.Y. He has a doctorate in computer science from George Mason University, master's degrees from the University of Illinois at Urbana-Champaign and the University of Southern California, and a bachelor's degree from Brigham Young University.

National Defense University
408 4th AVE
Fort Lesley J McNair
Washington, DC 20319
Phone: (202) 685-4325
Fax: (202) 685-4175
DSN: 325-4325
E-mail: alfordk@ndu.edu

How to Secure Windows PCs and Laptops

Terry Bollinger
The MITRE Corporation[1]

Progress brings new dangers: Powerful home computers, inexpensive high-speed Internet access, telecommuting, and software flaws have combined to create a soft underbelly in the defenses of government and corporate networks – and there is evidence that this weakness is being exploited. As of early 2005, many home and small business systems contain uninvited programs that capture keystrokes and passwords, hand control over to hostile users, and lull their owners into a false sense of security by undermining their virus and spyware checkers. The goal of such hardcore spyware programs is to help attackers profit at your expense by stealing your data, resources, money, or identity. Home PCs and laptops used in telecommuting are tempting targets for hardcore spyware since they provide easy access to data that otherwise can be found only in well-guarded enterprise networks. This article describes defensive strategies for reducing your level of risk from hardcore spyware.

Spyware is a form of uninvited malicious software that resembles viruses, but has a much more specific goal: Spyware steals your data, time, computer resources, or even identity so that someone else can profit from them. Because most anti-virus products prior to 2004 did not look for spyware, occurrences of it in home and small business Windows PCs and laptops have exploded. For example, at a recent demonstration I gave on how to remove spyware from Windows systems, a network engineer brought in a Windows 2000 laptop that his wife used in her small business. He kept it behind a firewall, one that was noted for its ability to block Internet viruses, and he had scanned it several times for spyware. Consequently, he did not expect to find any significant malicious software.

As it turned out, the laptop contained a keylogger, which is a type of hardcore spyware that records every password and keystroke. It also contained a transponder for recording entries into forms and installing other forms of spyware, and a hidden reconfiguration of standard Windows utilities that made the laptop into a Web server whenever it was online. The real shocker was yet to come. After removing the keylogger, tracker, and Web-hosting configuration, I clicked on its factory-installed virus checker. Immediately, the same hardcore spyware programs just removed reappeared and tried to install themselves. This time, however, the newly installed active spyware guards caught them. The spyware had been hiding in the standard-issue virus checker, ready to go the first time anyone tried to use the virus checker.

How could someone who had taken more-than-average care to secure his wife's laptop be so wrong about the degree to which it had been taken over by spyware? Was the takeover an aberration, some sort of one-in-a-thousand situation, or was it a symptom of a larger and more serious problem?

More Scary Stories

If you try applying all the spyware removal methods I describe in this article to home or small business Windows systems that belong to you and your friends, you will not have to take my word for how bad the spyware situation has become. You will find out for yourself. For now, let me give you a few more examples I have encountered of how bad the spyware situation has become:

- **Self-Destructing Computers.** Users who attach Windows systems to broadband Internet connections without a firewall can become infested with spyware so quickly that the system may become unusable within days. In one system that had been cleaned of spyware, the owner let his active spyware guards expire. By the end of the day it had stopped responding to anything the owner did, even while its disk and Internet connection kept churning furiously.
- **Massive Infection Rates.** A Windows system that has never before been checked for spyware typically has hundreds to thousands of the infections. While most of these will be the milder varieties that clog your system but do not steal your data, it is also common for such a system to have a few instances of the much more serious kinds such as keyloggers and remote administration terminals. Such massive rates of infection should fall rapidly throughout 2005 with the release of the new Microsoft AntiSpyware product, but only for the Windows 2000, XP, and 2003 systems to which it applies. Users of older Window 95, 98SE, and ME systems may even see an increase in spyware infection rates as attackers shift their focus to the more vulnerable older Windows systems.
- **Subverted Security Applications.** Here is another example of how virus checkers can be converted to the cause of protecting spyware. In late 2004, I uncovered a previously undetected keylogger and two remote administration terminals by using a new spyware checker, GIANT AntiSpyware. (Microsoft subsequently bought this application and renamed it Microsoft AntiSpyware.) However, although GIANT was able to find this previously undetected hardcore spyware, it could not remove it. GIANT would always seize up or, remarkably, de-list its finds before it reached the point where it could remove them. On a hunch, I temporarily uninstalled the huge, multi-function professional Internet security suite that I had been using for years. Sure enough, GIANT was able to finish removing the hardcore spyware. At some point, my Internet security package had been subverted and was being used to help the spyware hide and survive!
- **Outright Break-ins.** Imagine a Windows system that is invisible to any kind of probes from the Internet, arguably free of viruses, spyware, and subverted security applications, and resides behind two very tightly configured firewalls. How often would you expect someone on the Internet to break into such a system, scrounge around for interesting files, and send the results back to themselves? My own experience has been that it occurs somewhere in the range of once a week to once a month. Some of the files that I logged as they were being smuggled out included excerpts from public but hard-to-find Enron e-mails, registry settings for my speaker and microphone drivers, settings for one of my spyware tools, and an e-mail about a spyware article I had written. Ironically, the source of the break-ins later proved to be a security hole in the same data-log-

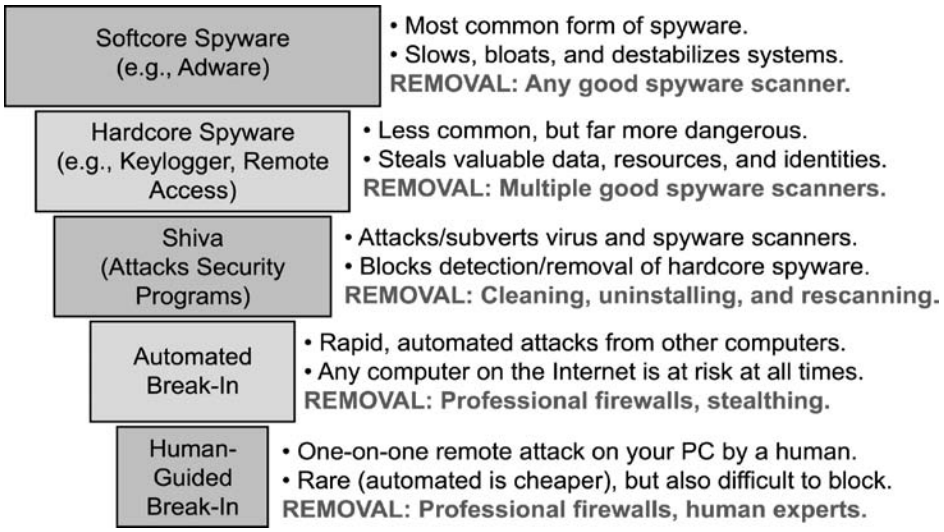


Figure 1: *The Spyware Hierarchy*

ging tool I was using to record the thefts! If you use your home or laptop systems for official-use-only government documents, patents, or company financial information, these kinds of subtle break-in routes should give you a good reason to worry.

The Spyware Hierarchy

To understand the dangers of spyware it is important to know the main varieties, which vary widely in how difficult they are to remove. The main types are shown in Figure 1. At the top of the spyware hierarchy is the most prevalent, least damaging, and easiest to remove spyware. *Adware*, which presumes to take over small chunks of your computer's resources to present ads to you whether you want them or not, is an example of this type of relatively mild or *softcore* spyware. When I first began looking for spyware on my home systems, adware and other types of softcore spyware typically outnumbered the more dangerous forms by ratios of hundreds or thousands to one.

Adware can be interpreted as an aggressive form of advertising, which is one reason why virus scanner companies made the unfortunate decision not to look for it or remove it when scanning systems for malicious software. The decision was unfortunate because adware has likely become the single most common cause of severely degraded performance in Windows systems. While a single adware program does not consume many resources, the problem occurs when marketers from around the globe descend over the Internet to engage in an adware feeding frenzy – with your computer as the main course. The resulting unintentional situation is surprisingly similar to what is known in cyber warfare circles as a *denial of service* attack, in which a computer or network is brought down by flooding it with trivial requests.

The next level down, *hardcore spyware*, is far more dangerous. This class of spyware makes no pretense of being legal, and so, unlike softcore spyware, it tries its best to stay hidden. Hardcore spyware includes keyloggers, remote access software, activity trackers, and in general anything that places your system and personal activities under the control of someone who has no business nosing around in them. Ironically, one reason why virus scanners were slow to look for hardcore spyware is that there are many legitimate software products that are legal if used strictly on systems you own and control, but which instantly become hardcore spyware if placed without permission on someone else's system. Other forms have no situations in which they are legal, and survive by techniques such as changing rapidly to avoid detection by scanner programs.

Sometimes, threats grow faster than the terminology used to describe them. A worrisome new trend is for hardcore spyware to be accompanied by helper programs that protect the spyware by going on the offensive against the very programs that are supposed to protect your system, such as virus and spyware scanners. My own name for this poorly recognized category of spyware is *Spyware HIV-like Attackers*, or *Shivas*, in reference to their unnerving ability to undermine the security software *immune systems* of Windows systems.

At present, spyware checkers do not scan for Shivas, although they can detect them indirectly by spotting their spyware clients and progeny. In the absence of tools to search for Shivas directly, the only way to find them is to look for unexpected protection or reinstallation of spyware. The only way to remove a Shiva is through a process of cleaning and uninstalling so that it keeps clearing out their hidey-holes until they become too damaged to function.

Beneath the Shivas are *automated break-ins*, which are the result of spyware on the Internet that is trying to get into your system. Automated break-ins upset the traditional wisdom that avoiding certain Web sites and not opening unrecognized e-mail attachments will keep you from getting malicious software. The sad truth is that if you do nothing more than attach a Windows PC to the Internet over a high-speed line, it will be subjected to the first stage of an automated attack – specifically, what is known as a directory services (port 445) query – within *seconds*. Automated attacks are why having a good firewall is so critical to keeping your system secure. Without good firewalls, automated break-ins can take over unguarded Windows systems in as little as a few hours or even minutes.

The deepest and most dangerous level of the spyware hierarchy is to get the attention not just of an automated attack program, but of a real, live human attacker who is well versed in the flaws of a wide range of Windows systems. Unlike an automated attacker, a human can apply creativity and ingenuity to breaking into your system, and so can be particularly difficult to keep out. Once they break into your system, a human attacker will typically place hardcore spyware and Shivas to ensure easier access to it in the future. The one positive side of human attackers is that in comparison to automated attacks, they are rare. There are simply not enough human attackers around to mount the same level of persistent attacks as automated methods can.

How Could This Happen?

How could spyware have gotten so bad without more recognition of the problem? A large part of what happened is that spyware evolved too rapidly from innocuous sources. Softcore spyware, for example, has its roots in ordinary advertising, which made it seem fairly harmless until it began choking systems with so many ads that the systems stopped working.

Similarly, several forms of hardcore spyware began their careers as self-monitoring programs intended to increase security, not undermine it. It was only when such programs started showing up unannounced on other people's systems that they became as dangerous as they are now.

Finally, and most worrisome, there has always been a well-hidden community of technically proficient but utterly unscrupulous attackers who are far ahead in their

understanding of the flaws and holes in Windows and other operating systems. This group does not announce its intentions; it simply takes the maximum possible advantage of any and every hole and weakness it finds.

Microsoft is taking the spyware threat very seriously. In November 2004, they purchased a company with one of the best products for finding dangerous spyware, GIANT AntiSpyware. Within two months they began distributing it for free to all Windows XP and 2000 users under its new name, Microsoft AntiSpyware. This purchase bodes well for the future, since it seems to indicate that Microsoft has recognized the seriousness of the spyware threat and is taking action to help reduce levels of infection in their XP and 2000 systems.

How to Secure Windows Systems

The seven major steps in securing a Windows system against spyware and related threats are given in the following sections. The first step, removing all spyware, is the most complex and important step in the overall process.

1. Remove All Softcore, Hardcore, and Shiva Spyware

This is the heart of the diagnosis and removal procedure, and often produces surprises. The three general techniques underlying the procedure are: (1) repeated removal of temporary files in which spyware typically hides, (2) use of multiple detection tools to increase coverage and confuse any Shivas that may be in the system, and (3) uninstall followed by a reinstall of any tool that appears to be malfunctioning. The strategy is one of attrition, in which you use your local advantage of physical possession of the computer to disrupt and confuse hardcore spyware and Shivas until they are too damaged to avoid detection.

The tools used in the removal process are all either free for home use or demos that can be used for initial cleanup even if you do not purchase them. Links to these tools, direct downloads of the entire set, and the latest detailed version of the spyware removal process can be obtained from my Web site at <http://terry bollinger.com>. Below is a very quick summary of the detailed spyware removal process that can be found and downloaded from my Web site:

1. (Optional) Prepare your computer:
   (a) Download the needed free and demo anti-spyware applications.
   (b) Backup any important data and applications.
   (c) Disconnect your system from the Internet.
   (d) Turn off all sharing of printers and folders.
   (e) Kill (end) all unnecessary Windows tasks.
2. (Optional) Uninstall unnecessary Web programs and virus checkers.
3. (Optional) Install Mozilla Firefox (free).
4. (Optional) Install Flash Player for Mozilla Firefox (free).
5. Install and use EasyCleaner (free) and remove TEMP files (Start>Run, type %TEMP%).
6. Install, update, and use CWShredder (free).
7. Install and use Sygate Personal Firewall (free for home use; Pro costs less than $50).
8. Install, update, and use SpywareBlaster (free).
9. (Unavailable for Win98 and ME users) Install and use Microsoft AntiSpyware (free).
10. Install, update, and use Webroot Spy Sweeper (30-day demo).
11. Install and use Spybot Search and Destroy (free).
12. Install and use Lavasoft Ad-Aware SE Personal (free for home use).
13. If any tools fail, uninstall them and then reinstall them.

2. Fully Update Windows and Internet Explorer

Once Windows has been cleared of spyware, Windows Update should be used to verify that all of the latest security updates have been downloaded for both Windows and Internet Explorer. Even if this was done earlier, it is a good idea to try it again in case spyware was interfering with the update process.

3. Uninstall Any Unnecessary Communication Programs

Even simple, practical programs such as time synchronization programs can cause problems, as can widely used auto-updating programs such as automated screen savers with thousands of available images. Multimedia programs, which often include their own softcore spyware, can also be problematic when used online. While harmless in themselves, such programs provide well-known forms of communication that a hacker can hijack and use for alternative purposes.

4. Make Sure Your System Is Stealthed (Invisible) on the Internet

Systems are stealthed, or hidden, on the Internet when they refuse to respond to queries to a number of standard port locations. To determine whether your system is visible, I highly recommend using the free *Shields Up!* Web site at <www.grc.com/x/ne.dll?bh0bkyd2>. Another very good site that also provides searches for Trojans is Sygate Online Services, <http://scan.sygatetech.com>.

5. Add Active Spyware Guards

You should keep at least one spyware guard active at all times. Spybot Search and Destroy provides a free spyware guard, TeaTimer. Microsoft AntiSpyware also provides a free spyware guard, but only for users of the later Windows 2000, XP, and 2003 operating systems. If you decide to purchase Webroot Spy Sweeper, it also provides active guards.

6. Maximize Firewall Protection

The choice of a firewall is the most critical one to bringing break-ins down to negligible levels. Based both on features and the success I have seen so far at cutting down residual break-ins to my own systems, my personal recommendation for this step is the Sygate Personal Firewall Pro. Other firewall options with good backing and reviews – not all of which I have tried myself – include ZoneAlarm Pro, Symantec's Norton Personal Firewall 2005, and McAfee's Personal Firewall Plus. All of these can also be purchased as parts of security suites. Due to the importance of your firewall for fending off attempts to break in to your system from the Internet, it is usually worth getting the most powerful firewall products available, which are generally called the *Pro* versions. Set it to the maximum protection, then experiment to find the least number of settings you must turn off to allow your applications to work correctly.

7. Make Sure Other Systems on Your LAN Are Similarly Protected

A common spyware tactic is to use the first PC captured in a home or small business network as a beachhead for launching powerful attacks against other computers in the same network. Since locally networked computers have special rights and higher bandwidth than remote computers operating over the Internet, such attacks can be far more difficult to fend off. For this reason, it is vital that all PCs and laptops sharing a LAN be made fully secure. An alternative is to completely isolate your PC from others in the local network, but that can be difficult to accomplish if you share a single DSL or cable Internet connection with them.

Coming Events

July 7-10
Internet, Processing, Systems, and Interdisciplinaries (IPSI) 2005
Cambridge, MA
http://internetconferences.net/usa2005/index.html

July 12-15
2005 IEEE Conference on Services Computing
Orlando, FL
http://conferences.computer.org/scc/2005

July 17-20
17th ACM Symposium on Parallelism in Algorithms and Architectures
Las Vegas, NV
www.spaa-conference.org

July 17-21
MobiQuitous 2005
The 2nd Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services
San Diego, CA
www.mobiquitous.org

July 21-26
CINC 2005
7th International Conference on Computational Intelligence and Natural Computing
Salt Lake City, UT
www.jcis.org/pages/subconference/cinc/cinc.aspx

July 24-29
Agile 2005
Denver, CO
www.agile2005.org

July 26-29
CMMS 2005: The Computerized Maintenance Management Summit
Indianapolis, IN
http://maintenanceconference.com/cmms

May 1-4, 2006
2006 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

Conclusion

When it comes to small computer systems linked to a global Internet, the days in which we could fully trust the results of our own computers and applications are gone. Whether we like it or not, there are people and communities on the Internet who view home users and their systems as prey – and they will go to remarkable lengths to capture such prey.

Making home and small business Windows systems more secure not only prevents you from becoming the victim of such a network predator, but also adds to overall security for everyone by removing the ability of your system to spread spyware, and by reducing the total quantity of sensitive data that can be collected on a given topic. While the procedures given here require some effort, I think it is fair to say the benefits are well worth that effort.◆

Note

1. This article is the result of independent work by the author, and does not necessarily reflect the views of his employer, The MITRE Corporation, which is listed here for identification purposes only.

About the Author

Terry Bollinger works at The MITRE Corporation, McLean, Va., on acquisition of leading-edge information technologies. He is an Institute of Electrical and Electronics Engineers (IEEE) Millennium Medal Winner, a former assistant editor-in-chief for *IEEE Software*, the main editor for the recent Nov/Dec 2004 special issue of *IEEE Software* on Persistent Software Attributes, and the author of a widely quoted 2002 survey of how open source software is used in the Department of Defense. His personal Web site, <http://terrybollinger.com>, provides detailed information and tool downloads for removing spyware from Windows computers.

Phone: (703) 588-7410
Cell: (703) 309-6317
Fax: (703) 588-7560
E-mail: terry@mitre.org

Web Sites

PC World
www.pcworld.com/downloads/browse/0,cat,1727,sortIdx,1,pg,1,00.asp
PC World editors present a review of their favorite anti-spyware tools in this online article, including Ad-Ware SE v1.05, Spybot Search and Destroy v1.3, CWShredder v2.1, Microsoft Anti-Spyware Beta, and NoSpyMail. The site also features the article "Avoid Identity Theft: Best Files to Protect Your Privacy."

iSecuritySource.com
www.isecuritysource.com
The iSecuritySource.com Web site features details and information about the spyware detected on your computer from scanning, including Trojans, worms, viruses, adware, and dialers, and provides removal instructions. The site also features an "Ask the Experts" forum.

Home PC Firewall Guide
www.firewallguide.com
The Home PC Firewall Guide provides easy access to basic information about, and independent third-party reviews of Internet security and privacy products for home, telecommuter, and small office/home office end-users.

PCMag.com
www.pcmag.com/article2/0,1759,1758380.asp
PC Magazine's Web site notes that the majority of PCs are infested with spyware. The explosion of dedicated anti-spyware applications in the past year began to address the growing problem; this article reviews eight of the leaders.

Microsoft Windows AntiSpyware
www.microsoft.com
Windows AntiSpyware (Beta) is a security technology that helps protect Windows users from spyware and other potentially unwanted software. Known spyware on your PC can be detected and removed, helping reduce its negative effects, including slow PC performance and annoying pop-up ads. Continuous protection improves Internet browsing safety by guarding more than 50 ways that spyware can enter your PC.

How and Why to Use the Unified Modeling Language

Lynn Sanderfer
TecMasters, Inc.

This article addresses the Unified Modeling Language and its purpose, constructs, and application to defense software development applications.

The Unified Modeling Language (UML) is a notation that can be applied to the software development process. UML, in itself, is not a software development process, but a system modeling language developed by Grady Booch, James Rumbaugh, and Ivar Jacobson of Rational Software. It provides a design notation whereby the scenarios for which the system requirements can be utilized are depicted as well as subsequent design notation for the chosen implementation. This methodology was a merging of these men's separate practices in object-oriented (OO) software design notation[1]. The primary goal of UML is to model systems using OO software (also referred to as architectural-based software).

UML: The Why

Whenever something is built, drawings are made to describe the look and feel of the entity being built. These drawings work as a specification of how we want the finished product to look. The drawings are handed over to *builders* or are broken into more detailed drawings necessary for the construction. A well-architected product pays off in the end, but high quality does not just happen. Software quality is a result of correctly understanding the requirements, a solid design that is readily implemented into code, quality user documentation, and is committed to satisfying the user's needs.

Software design is the equivalent of these construction drawings, and is developed simply to produce a software solution to a problem. The process of software design can be described as an activity in which the designer develops a highly abstract model of a solution and then transforms it into a very detailed design. OO, or architectural-based design, guides the designer into thinking about the decomposition of problems into a collection of autonomous agents or objects that can be mapped as closely as possible to a physical representation of the system. The designer can then resolve the system in terms of behaviors and responsibilities of objects.

By reducing the interdependency among software components, architectural-based programming permits the development of reusable software. Such software components can be created and tested as independent units in isolation from other portions of a software application. Programming then becomes the simulation of the model spectrum. Keep in mind that the designer's ultimate requirement is to meet the *fitness of purpose* of the user's needs: Does it work and does it do the required job as well as possible[2]?

Note that it takes less time to build a system by making *instructions* (and following them) than it would take to start from scratch to build a system without directions. This is because documenting the specifications to the desired product allows the analyst/developer to (1) verify his understanding of the task at hand, (2) visualize and identify the most crucial components of the system, and (3) identify goals that keep the team focused to the system objectives. The time considered gained by omitting the development of concrete plans is paid for many times over in misinterpreted requirements, inefficient code implementation, faulty software, and unhappy end-users.

UML is a standardized design language that provides a mechanism for creating this design, particularly when the software being developed is done by using OO principles.

UML uses foundation pieces that describe the abstraction of system components called classes, and the instantiation of those classes into system objects that are later used to provide management of component functionality and data. For further details on OO techniques, see the *Notes* section at the end of this article.

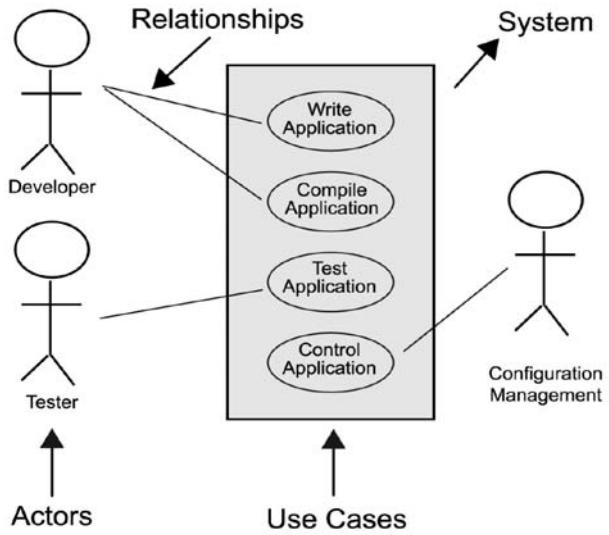
UML: The How

UML is defined by depicting the software from various aspects or views. Each view can be displayed using a variety of diagrams that detail the contents of the views. Each diagram is composed of OO concepts to include classes, objects, messages, relationships, and dependencies. These concepts or elements are persistent throughout the design process, or rather they do not change meaning or symbology throughout the design.

By looking at a physical system from different views, a developer or user can concentrate on one aspect of the system at a time. The UML views are the following:

- **Use Case View:** Depicts the functionality of the system as perceived by the external actors.
- **Logical View:** Depicts how the functionality is designed inside the system.
- **Component View:** Depicts the organization of the code components.
- **Concurrency View:** Depicts concur-

Figure 1: *Use Case Diagram*

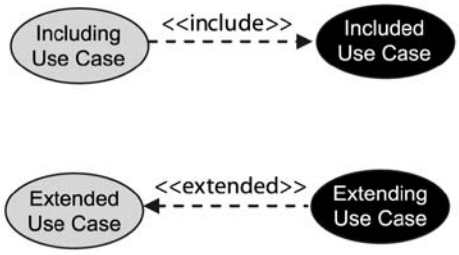


rency in the system, addressing the problems with communication and synchronization that are present in a concurrent system.

- **Deployment View:** Depicts the deployment of the system into the physical architecture with computers and nodes.

To display these views, UML introduces nine different types of diagrams: Use Case Diagrams (see Figure 1), Class Diagrams, Object Diagrams, Sequence Diagrams, State Diagrams, Collaboration Diagrams, Activity Diagrams, Component Diagrams, and Deployment Diagrams.[3]

Figure 2: *Extended/Include Use Case Notation*



Use Case Diagrams

Use case modeling is a design modeling technique that uses both verbal and graphic descriptions to reveal what a new system should do or what an existing system already does via Use Case Diagrams. These diagrams are the starting point when designing a new system using UML. This notation or diagram is designed to communicate exactly what is expected of a system upon completion to management, customers, and other interested parties.

There are four basic components of Use Case Diagrams:

- System.
- Actors.
- Use cases.
- Relationships.

The system is the entity that performs the function. Actors are entities, people, or other systems that use the system to be developed. Use cases are the actions that a user takes on a system. Relationships depict how actors relate to use cases.

If additional functionality is required in a use case description, this can typically be handled by *include* and *extend* relationships (see Figure 2). The include relationship is used to indicate that a use case will *include* functionality from an additional use case to perform its function. Similarly, the extend relationship indicates that a use case may be extended *by* another use case.

To describe use cases, a software designer would first identify the actors of the system. Next, designers would ask the providers of the requirements for more information about what they want. A designer should always keep these questions in mind as this information exchange progresses: "What will the system do?", "What input/output does the system need?" and "What are the major problems with the current implementation of the system?"

Figure 3: *Sample Class Diagram Notations*

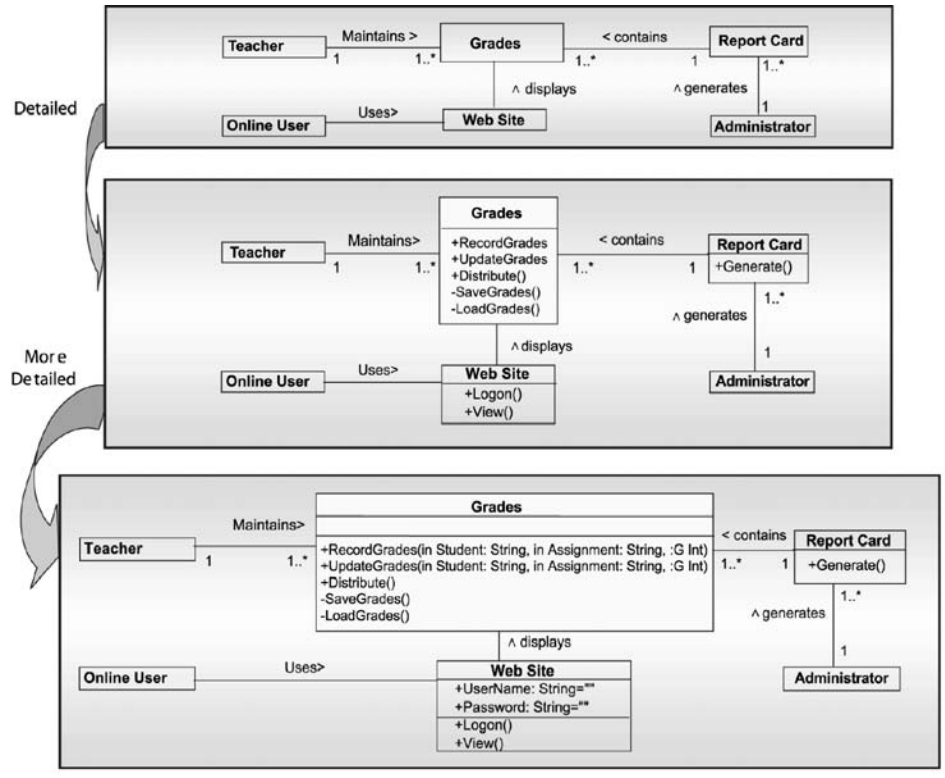
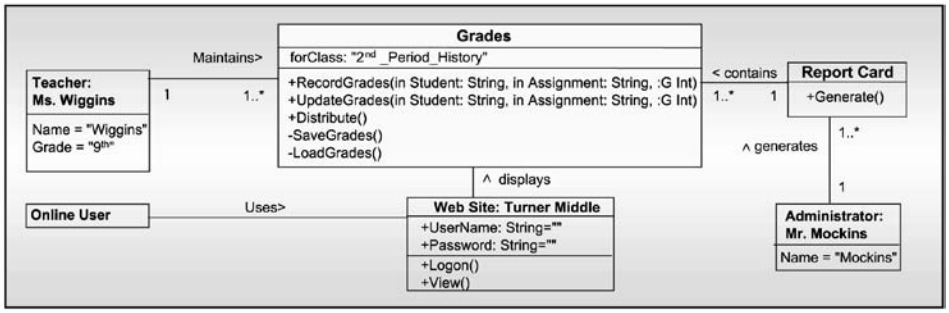


Figure 4: *Sample Object Diagram*



Class Diagrams

A Class Diagram describes the static view of the system (see Figure 3). It is a functional representation of the system that reveals where data resides and where functionality is available through the use of class attributes and operations to outside classes. Class Diagrams define the foundation for other diagrams, and show classes of the systems and relationships among the classes. If object instances of a Class Diagram are shown, it becomes an Object Diagram (see Figure 4).

To create a Class Diagram, the classes have to be identified and described. A class is drawn with a rectangle and divided into three compartments: the name compartment, the attribute compartment, and the operation compartment. The name compartment of a Class Diagram contains the name of the class. It is typed in boldface and centered. The attribute compartment contains the characteristics that describe the class. For example, a computer class would have manufacturer, model number, storage size, speed, and perhaps operating system as attributes. The operation compartment contains the operations or methods by which the attributes are manipulated, as well as other system functionalities. The operators in a class describe what the class can do. Both attributes and operations of a class can have different visibility (+ for public, - for private) privileges. Classes show their relationships by way of associations or a semantic connection between objects that

indicate the direction of service, *how many* relationships called multiplicity, and repetitiveness.

Sequence Diagrams

A Sequence Diagram describes the dynamic view of the system (see Figure 5). This diagram is important in that it shows the sequence of messages sent between the objects with respect to time. The Sequence Diagram consists of a number of objects shown with vertical lines. The objects are activated from left to right with an indication of the exchange of messages between the objects. Messages themselves are shown as lines with message arrows between the vertical object lines.

Each object in a Sequence Diagram is represented by an object rectangle with the object/class name underlined. A vertical dashed line down from the object, called the object's lifeline, indicates the object's execution during the sequence. Communication between objects is represented as horizontal message lines between the object's lifelines. The arrows indicate whether the message is synchronous (flow is interrupted until the message has completed), asynchronous (active object does not wait on a response), or simple (flat flow depicting control is passed without indicating details). Sequence Diagrams can also show branching, iteration, recursion, creation, and destruction of objects.

State Diagrams

State Diagrams are dynamic in nature and depict how an individual object changes state when a behavior is invoked (see Figure 6, next page). It also indicates the invoking event. A state diagram should be attached to all classes that have clearly identifiable states and complex behavior.

A state-chart diagram is composed of states, transitions, and events. A basic state is shown as a rectangle with rounded corners. The name of the state is placed within the rectangle. The first state in a model, or the start state, is simply a solid dot. The last state in a model (the end state) is a solid dot with a circle around it. Transitions are used to show flow from one state to another. A transition is modeled by an open arrow from one state to another.

Collaboration Diagrams

Collaboration is another depiction of the dynamic behavior of a system (see Figure 7, next page). Collaboration Diagrams, showing both a context and an interaction, can be thought of as a combination of the Class Diagram and Sequence Diagram. Collaboration Diagrams actually model either objects or roles and their sequenced communication between each other. Message sequence in Collaboration Diagrams is identified by numbering the messages with labels.

Message type in Collaboration Diagrams is the same as in Sequence Diagrams: synchronous, asynchronous, and simple. Messages sent in parallel can be described using letters in the sequence number expression. For example, the sequence numbers 1.1a and 1.1b of two messages in the collaboration diagram indicate that those messages are sent in parallel.

Activity Diagrams

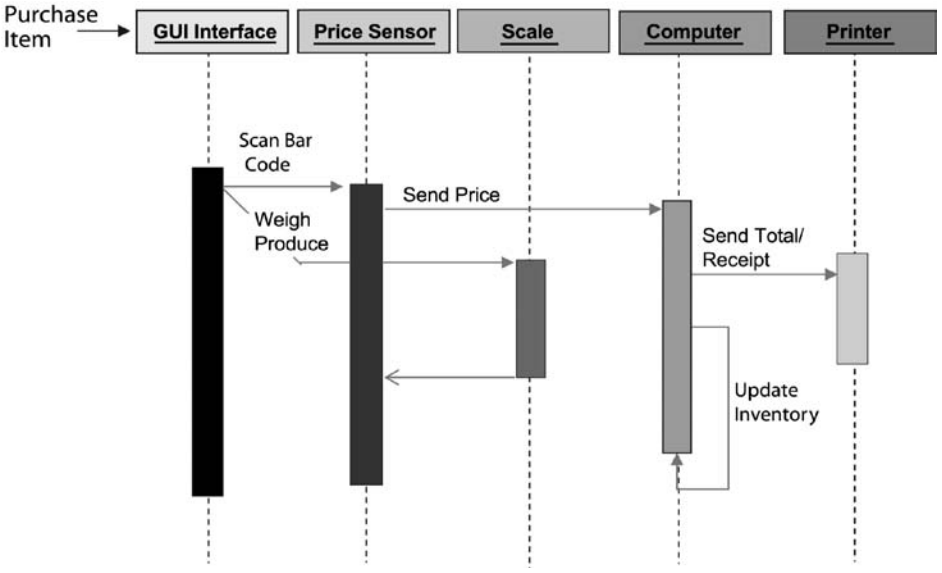
The Activity Diagram captures actions as the states of a system change (see Figure 8, page 17). Activity Diagrams focus on work performed in the execution of a function. The states in the Activity Diagram transition to the next stage directly when the action in the state is performed without specifying any event. Activity Diagrams also have swimlanes, or a grouping of activities according to the responsible software entity. Activity Diagrams show the following:

Object-Oriented or Architectural-Based Concepts

Architectural-based design models the problem in terms of a set of particular entities or objects that can be recognized in the problem itself, together with a description of the relationships that link these entities. The initial description of the system is defined in an abstract top-down process. A complete and detailed model of the solution is then developed by elaborating the descriptions of the entities and the interactions occurring between them. The strategy then is composed of grouping elements together in the design that can be described with the same functionality. The implementation is done as a bottom-up development process. Programming from an architecturally based design is comprised of the following concepts:

- Object is the encapsulation of data values (or states) and operations (or behaviors) into one executable entity.
- Class is the abstract or generic description of a concept in terms of data types and operations.
- Inheritance is the principle that knowledge of a more general category is applicable also to the more specific category. In this way classes can be organized into a hierarchical inheritance tree.
- Information hiding is the principal by which a client sending a request or invoking a process need not know the actual means by which the request will be honored.
- Abstraction is the ability to describe a type or functionally in generic terms thereby isolating design and execution information. Information hiding is a specific type of abstraction.

Figure 5: *Sample Sequence Diagram*



- The work that will be performed when an operation or method is executing.
- The internal work of an object.
- How a set of related actions may be performed, and how they affect objects around them.
- An instance of a use case in terms of object state changes.

Activity Diagrams can have a start and an end point. A start point is shown as a solid filled circle; the end point is shown as a circle surrounding a smaller solid circle. The actions in an Activity Diagram are drawn as rectangles with rounded corners. Within the action, a text string is attached to specify the action(s) taken. Transitions between actions are shown with an arrow, to which guard-conditions, a send-clause, and an action-expression can be attached. A diamond shaped symbol is used to show a decision point. Swimlanes group activities, typically, with respect to their responsibility. Swimlanes are drawn as vertical rectangles. The activities belonging to a swimlane are placed within its rectangle with a name at the top.

Component Diagrams

A Component Diagram is a type of implementation diagram (see Figure 9); it shows where the physical components of a system are going to be placed in relation to each other. In the Component Diagram the physical *pieces* are the actual software entities.

A component is shown in UML as a rectangle with an ellipse and two smaller rectangles to the left. The name of the component is written below the symbol or inside the large rectangle. A software dependency is shown as a dashed line with an open arrow and indicates that one component needs another to be able to have a complete definition.

Deployment Diagrams

The deployment view is the second type of implementation diagram and shows the physical layout of the software system after being installed on the actual hardware system and how the software components will interact with each other. Deployment Diagrams (see Figure 10) are described in terms of nodes, connections, and software components executing on nodes.

Nodes are physical objects that have some kind of computational resource. This includes computers with processors, as well as devices such as printers, communication devices, card readers, and so on. A node is drawn as a three-dimensional cube with the name inside it.

Nodes are connected via a communication association. The communication type is represented by a stereotype that identifies the communication protocol or the network used. Executable component instances may be contained within node instance symbols, showing their physical residence and execution on the hardware.

Figure 6: *State Diagram Example*

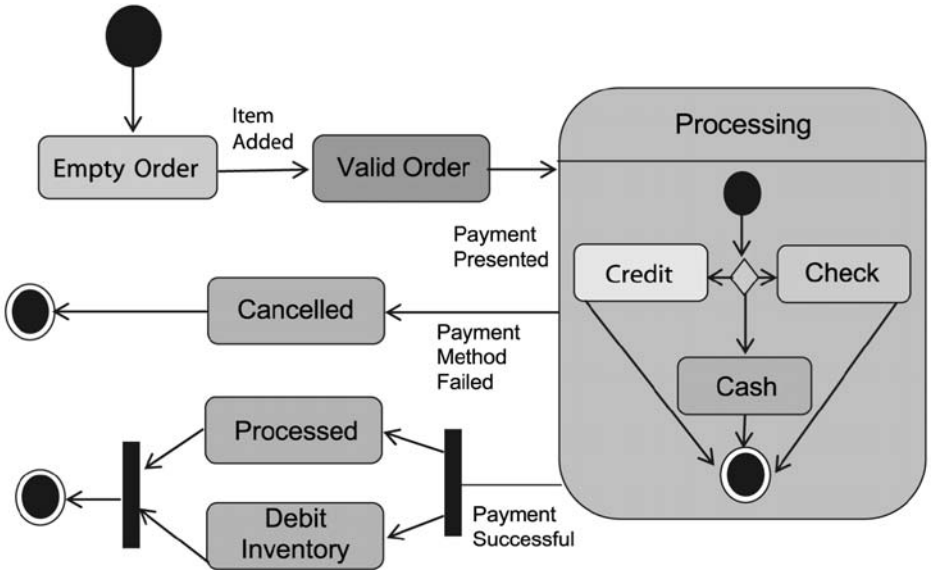
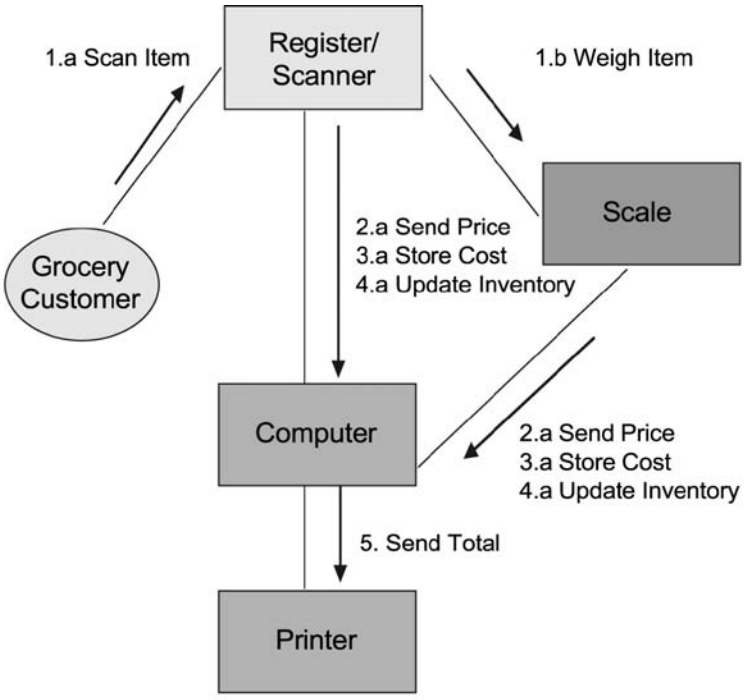


Figure 7: *Collaboration Diagram Example*



Conclusion

UML is a standard practical methodology used in representing OO systems, regardless of the procurement or financial aspects of the program, i.e., commercial or government. Advanced software projects or projects that are committed to leading-edge software technologies and concepts typically find value in designing their software systems to the physical systems that they represent. This allows non-software personnel to grasp the concept of operations of the system without having to understand software terminology. This also lends itself to reducing software life-cycle costs because it produces modular code with well-defined interfaces.

Historically, it can be shown that the more time rendered on requirements and design, the better the product when in the code and test phase. It does, in fact, improve the quality of your product, as well as other development factors, when the design of the system is clearly con-

veyed with respect to the intended requirement. However, as Doug Rosenberg stated [1], "The cold, hard reality of the world of software development is that there is simply never enough time for modeling."

The methodologies provided by UML are very rarely utilized exhaustively. There is simply not enough time or money to do so. In fact, I have seen excellent defense software efforts that implemented OO design with design techniques other than UML. However, to the designers and developers of software systems, UML offers very sound techniques in describing systems. Furthermore, if you are new to OO software development, UML provides a road map by which, if you take the time to depict the design in the various UML software design views, it is easier to produce a good product than not!◆

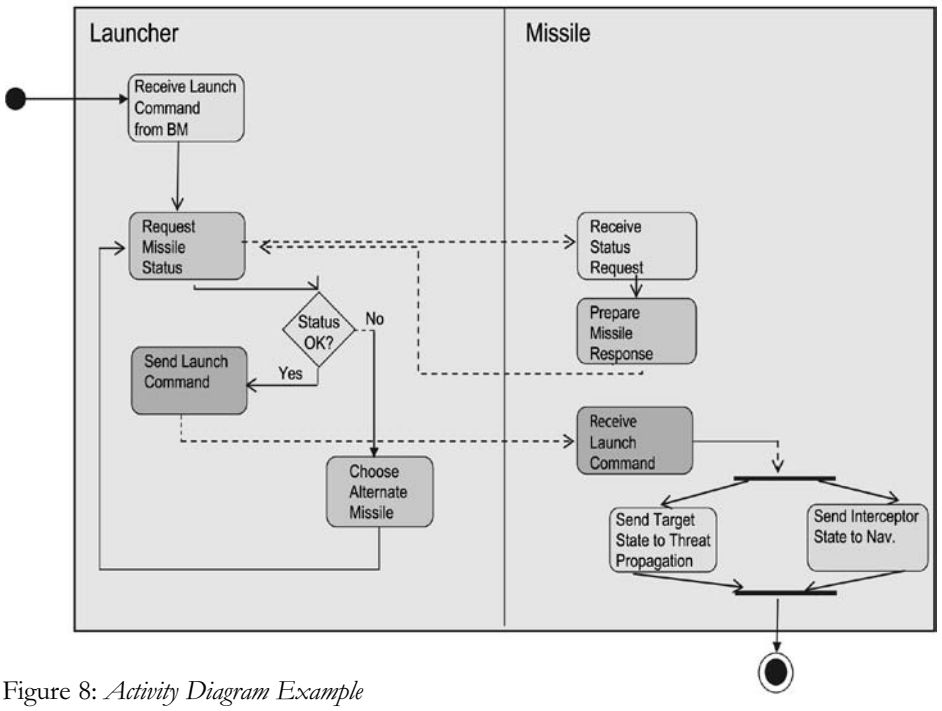


Figure 8: *Activity Diagram Example*

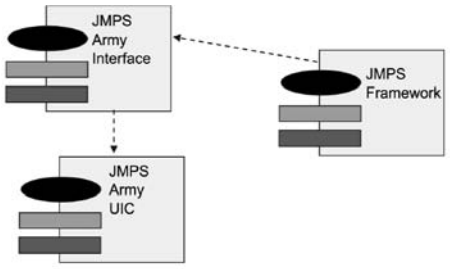


Figure 9: *Component Diagram Example*

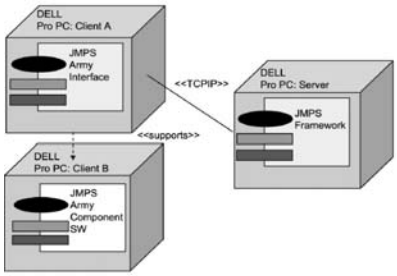


Figure 10: *Deployment Diagram*

Reference

1. Rosenberg, Doug, and Kendall Scott. Use-Case Driven Object Modeling With UML: A Practical Approach. Addison-Wesley Professional, Mar. 1999.

Notes

1. Grady Booch developed Booch diagrams. Booch defined the notion that a system is analyzed as a number of views where each view is described by a number of diagrams. The principals in his methodology were sound, based on depicting several views of the software; however, the Booch diagram notation that was based around clouds symbology was cumbersome to implement.
2. James Rumbaugh developed the Object Modeling Technique while employed at General Electric. His methodology was founded in expressing the software in various methods: the object model, the dynamic model, the functional model, and the use-case model.
3. Ivar Jacobson developed the Object-Oriented Software Engineering/ Objectory design methodology. His system is based on use-cases, which define the initial requirements on the system as seen by an external actor.

Additional Reading

1. Roff, Jason T. UML A Beginner's Guide. McGraw-Hill, 2003.
2. Eriksson, Hans-Erik, and Magnus Pinker. UML Toolkit. John Wiley & Sons, Inc., 1998.
3. Kroll, Per, and Philippe Druchten. The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP. Addison-Wesley, 2003.
4. Budgen, David. Software Design. Addison-Wesley, 1994.
5. Budd, Timothy. An Introduction to Object-Oriented Programming. Addison-Wesley, 1991.
6. Pressman, Roger S. Software Engineering: A Practitioner's Approach. 3rd ed. McGraw-Hill, 1992.
7. Stephen Prata. The Waite Group's C++ Primer Plus. 3rd ed. Indianapolis, IN: Sam's Publishing, 1998.

About the Author

Lynn Sanderfer is a software analyst for TecMasters, Inc. and is currently contracted on the Army Mission Planning Software Program. She has been in software development/software engineering for 18 years. Sanderfer has completed the Software Life Cycle Development Certification, the Software Engineering Management Certification, and the Advanced Software Development Certification offered by the Air Force Institute of Technology Software Professional Development Program. She is also certified as a Capability Maturity Model® Integration Auditor with CSSA, Inc., a licensed partner of the Software Engineering Institute. She has a bachelor's degree in engineering from the University of Alabama in Huntsville and is currently working on her master's degree in software engineering there.

TecMasters, Inc.
1500 Perimeter PKWY
STE 200
Madison, AL 35758
Phone: (256) 830-4000
Fax: (256) 830-4093
E-mail: theresa.sanderfer@us.army.mil

Effective Practices for Object-Oriented System Software Architecting

Rich McCabe and Mike Polen
Systems and Software Consortium

Development programs for software-intensive systems are increasingly attempting to employ object-oriented (OO) techniques and technologies – including OO design, the Unified Modeling Language (UML), and UML-based modeling and software development tools – in expectation of achieving greater flexibility, evolution, and productivity. However, these programs frequently experience a number of challenges when they insert OO design into their traditional practices. Unless both development organizations and acquisition offices make a thoughtful transition to OO design, they are likely to experience difficulties that may well endanger the anticipated benefits. This article describes some typical pitfalls of OO development and recommends a number of architectural practices that will help programs avoid or mitigate these dangers.

Object-oriented (OO) design as a software programming technique has been around almost 40 years, although it has become prominent in military and government systems development in only the last few years. Its use has grown from just a programming trick to a complete approach to analyzing and solving complex problems.

As with any technology, OO design has not always been applied successfully. Literature is full of misapplied OO techniques as well as impractical recommendations. The authors have seen both good and bad OO programming in large military systems. This article will try to enumerate the most common pitfalls and offer practices that have been shown to work.

The practices recommended in this article are motivated by the desire to use OO technology to its best effect. OO techniques provide the opportunity to capture the logic of a complex domain or problem environment in a structure that reflects or is congruent with interrelationships among actual domain elements. This structure encapsulates the impact of volatile requirements and design decisions, and provides a natural mapping from requirements to design. Consequently, the design is flexible, facilitating system integration and repeated adjustments or enhancements to the implementation during initial development and subsequent maintenance. Therefore, these attributes of congruence and flexibility reduce system lifecycle costs and are worth striving for rather than settling for any design that can be made to *work*.

Practices the authors have used to help struggling OO projects include the following:

- System-wide software architecture.
- Layered software architecture with domain layer.
- Goal-directed black box use cases.
- Spiral design.
- Architecture and code iterations.
- Spare modeling.
- Experienced OO guides.

These practices have been widely recommended by many OO experts. The authors' experience is that they are just as valid for complex system development within government acquisition programs. The authors have incorporated many of these recommendations into a prescriptive methodology [1].

> *"OO techniques provide the opportunity to capture the logic of a complex domain or problem environment in a structure that reflects or is congruent with interrelationships among actual domain elements."*

The following sections describe these practices in more detail. However, to keep this article within an acceptable length, it is assumed the reader is familiar with typical development practices that are referenced during the discussion.

System-Wide Software Architecture

One of the most effective means of preserving congruence between the domain and the design is to use OO techniques to decompose the software across the entire system. Working from a software perspective with such a broad scope yields a unified, system-wide, software architecture, expressing domain concepts with semantic consistency throughout the software.

Too often, programs follow past patterns of system design that preclude good OO design. System designers, too, readily define logical design components to match organizational structures, physical elements of the design, or sub-functions of deep functional analysis performed without sufficient feedback (e.g., any) from software design. While these decompositions may match conveniently to the specialties of different groups, correspond to the administrative hierarchy of the development team, or enable easy traceability, they constrain the scope of any software perspective to individual *boxes*, conflict with OO structuring, and increase the difficulty of maintaining overall semantic consistency (e.g., consistent interpretation of data) among the components.

Semantic consistency among different parts of a system is extremely important. Trying to maintain that consistency by merely matching interfaces of system components is not sufficient, especially if the major logical interfaces are forced to align with physical interfaces. OO approaches need to work with the software as a whole to define major class interfaces for flexibility and congruency, without the imposition of other decompositional schemes.

The only way to preserve flexibility and congruence of software design in balance with other tradeoffs is for OO developers to work closely with other team members during multiple iterations

of the requirements and design. Although software is malleable enough to fit into almost any decomposition, taking for granted the benefits of a system-wide OO structure is almost certain to compromise them.

Layered Software Architecture With Domain Layer

Grouping related classes into *layers* is another technique for managing and coordinating many classes in large systems. The term *layer* is something of a misnomer in that the layers are not strictly arranged like a cake. However, the classes in a layer should all deal with some common aspect of the design and have limited access to other layers (upper layers call upon the services of lower layers but not vice versa). Figure 1 is a representative example. Usually, only one or a few key classes present all the services of the layer to other layers, the other classes of the layer are hidden behind this interface.

One of these layers should be focused on the domain specifics of the system and be the key to understanding the domain. This layer is typically called the *domain layer* (or system or business logic layer), or is named for the particular domain (e.g., accounting, or battle management). Classes in this layer are defined to represent the essential concepts and relationships in the domain in terms that the subject matter experts understand [2]. It should have no knowledge of the underlying hardware, communication protocols, operating system features, or other aspects of the design known to other layers. The domain layer is key to the overall understandability and flexibility of the design.

Typical subsystem partitions are not equivalent to OO layers and usually chop up what would correspond to the domain layer. As discussed in the previous section, the design is often prematurely split into traditional subsystems before any consideration is given to overall domain congruence or flexibility. Frequently, the authors have seen designs structured in the form of all-knowing device-centered subsystems, each containing bits of domain knowledge intertwined with hardware interfaces and other types of design knowledge, and each interconnected to all the others. This kind of design arises from trying to scale up a simple data pipeline or *thread* to a system with multiple, interconnected data pipelines. The result is a fragile, inflexible design with bits of domain and design knowledge expressed in multiple locations.

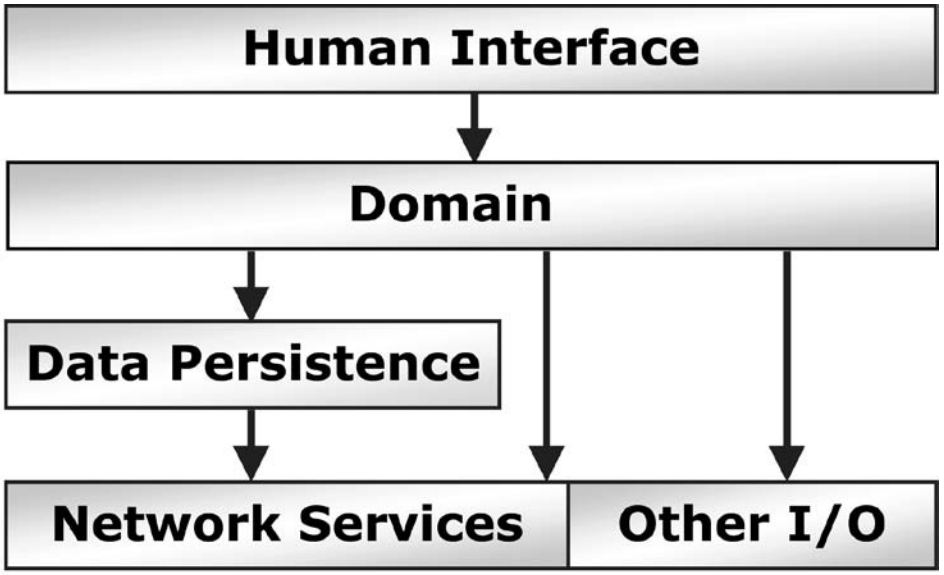


Figure 1: *Generic Layers*

Goal-Directed, Black Box Use Cases

The use case technique [3], when applied appropriately, has the advantage of clarifying the intent of system stakeholders without unduly presuming the design. Use cases are a narrative of system interaction with external entities. Best practice structures each use case around a single goal of a system user or stakeholder, and treats the system as a black box with observable behaviors. Although use cases are a technique for requirements analysis, their black box orientation prevents design assumptions from creeping into requirements. Furthermore, the emphasis on system interaction with external entities in meeting goals helps OO developers identify essential entities and concepts in the domain (as discussed in the previous section).

> *"Too often, programs follow past patterns of system design that preclude good OO design."*

Like any technique, use cases can be misapplied as in the following:

- Although use cases can be applied at different levels (notably to analyze systems containing people: at the outer level to express system interactions with external actors, and at an inner level to express interactions of the hardware/software with people within the system), it is a mistake to mix levels.
- Ignoring stakeholder goals as the organizing principle for use cases leads to haphazard narratives that do not clearly reveal how the system produces value for the stakeholders.
- Another common mistake is failing to abstract away from details of the interfaces, data, or interaction protocols in the use case narrative, or otherwise attempting to develop complete use cases that cover every possible detail or scenario. This treatment leads to a proliferation of endless use cases of rapidly diminishing value.

Use cases should be employed judiciously to illuminate key system behaviors, not to exhaustively document requirements. The opposite extreme, treating use cases as no more than the Unified Modeling Language (UML) use case diagram, is pointless.

Often, use cases are ignored entirely. True, use cases need to be augmented with other requirements techniques, but use cases provide a unique and important perspective.

Spiral Design

A spiral approach to OO design generally yields the best results. Here, a spiral approach means that the designers begin by addressing just a few key issues in an initial design, and then incrementally address other concerns and complications in multiple revisions. Initially, developers make only a rough allocation

of responsibilities to components, and gradually resolve the details as design issues are introduced and decided. Here is a partial ordering of issues that appears to work well:

- Domain congruence and flexibility.
- Mapping to physical architecture.
- System attributes.
- Concurrency.

Of course, program-specific circumstances impact this general scheme such as whether the development team has previously developed OO designs for similar systems.

Many programs attempt to directly create a design that simultaneously addresses all system issues and provides detailed interface descriptions as well. This invariably leads to an overly complicated design because the designer has too many concerns to juggle at once. The mental overload causes many errors and results in a *big ball of mud* [4]. Instead, the designers need to start with a simple yet admittedly inadequate design and work in complications one at a time, rebalancing earlier design elements as necessary.

Domain Congruence and Flexibility

The initial design captures the essentials of the domain to ensure flexibility and understandability (see previous section "Layered Software Architecture With Domain Layer"). The intent is to preserve the flexibility of the initial design as much as possible, but compromise it where necessary to address other demands.

Mapping to Physical Architecture

Multi-processor architectures potentially introduce a number of complications best delayed until the domain essentials have been identified. Assignment of functionality to various processors has subtle implications for both timing and reliability. Trying to force these decisions too early before more information is available about other aspects of the design (and even the implementation) is both difficult and dangerous, and tends to emphasize physical interfaces over logical (domain) interfaces. Delaying such decisions is usually advantageous, especially when facilitated by infrastructure technologies such as Common Object Request Broker Architecture (CORBA).

Similarly, OO designs typically encapsulate hardware interfaces inside classes, protecting the rest of the system from volatilities in the device interfaces. This technique usually relegates many of the decisions in *allocating requirements to software or hardware* as a secondary concern.

System Attributes

Timeliness, reliability, safety, security, and other such system attributes are difficult to address individually and often have tricky interdependencies. Designers tend to begin with their greatest concern (e.g., critical timing paths) and orient the design toward that aspect. In the authors' experience, designing for flexibility first and adjusting for timeliness as proves necessary is much more effective than designing first for timeliness and subsequently for flexibility. Similarly, designers need a preliminary design combining both hardware and software before they can meaningfully analyze the impact of other attributes.

"In the authors' experience, designing for flexibility first and adjusting for timeliness as proves necessary is much more effective than designing first for timeliness and subsequently for flexibility."

Concurrency

Concurrency decisions are another area best left until later. Concurrency can add tremendous design complexity. Start with as few concurrent elements that will possibly work (one is often the best starting point). Add new concurrent elements only after a performance test or real-time analysis has shown that the implementation will not work. The authors have seen a narrow design focus on timeliness lead to a (unsuccessful) system design with more than 100 concurrent elements on a single processor.

Architecture and Code Iterations

Although architectural analysis is important, systems today are too complex to rely solely on analysis to ensure that a design will exhibit the expected attributes. The flexibility of a design hinges on too many details that only become apparent with coding. Yet these coding details can have implications for the larger design.

Fortunately, software is well-suited to evolutionary development. Where testing and configuration management discipline is continuously applied, multiple iterations of design, code, and test (in parallel with deepening requirements analysis) are more productive and effective than a waterfall process. Coding the most critical or highest-risk portions of the design validates the solution approach. A robust, flexible design is discovered and becomes increasingly stable through multiple iterations. This practice also fits well with spiral design.

Even though the waterfall process is rarely, if ever, suitable to manage the risks of complex system development, it is still prevalent in government contracting. With the recent release of Department of Defense 5000 [5], the government is trying to rectify the traditional bias toward waterfall-planned programs, but the waterfall process remains predominant. Typically, teams doing OO development today within a waterfall process spend their time creating, editing, and reviewing UML diagrams. Unfortunately, translating from UML diagrams into code is not automatic and often exposes major flaws in the design.

When designers are inexperienced in both OO programming and the underlying infrastructure (such as CORBA) the resulting designs are often misguided or simply infeasible to implement. A common design defect is attempting to manage classes with a large number of objects under tight timing considerations. The kind of schedule pressure created by the emphasis of a waterfall process on getting everything right the first time leads developers to opt for the *most expedient fix* in code, rather than rethinking the design.

Generally, programs, as reflected in their plans and practices, do not appreciate just how much good design depends on feedback from prototypes, or preferably, from early implementations of partial or simplified designs. As program schedules become more compressed, development teams are, in fact, coding and designing simultaneously. However, rather than plan for rapid design and code iterations in an open and well-managed fashion, they often attempt to mask this reality beneath a simplified, waterfall model being projected for the program

and, consequently, create chaos rather than success.

Spare Modeling

Models can be a very powerful tool in working on a complex problem. Formal models (see [6]) can support automated checks for logical consistency and automated generation of effective test suites. Visual representations can contain rich semantics that are hard to convey in words alone. When used judiciously, they are of great value as an aid to communication among developers, especially when generated quickly, informally, and cheaply.

However, the idea that a system should be *completely* modeled using UML diagrams is of dubious value, if not outright harmful. The cost of developing and maintaining an extensive set of UML diagrams for a system far outweighs its benefit. The UML has been justifiably called a *cartoon* [7] in that it is not semantically sufficient to address all the nuances that must be communicated to a tester or coder. Development teams find it far too easy to expend indefinite effort on UML diagrams of arbitrary detail without any assurance that these diagrams connect to implementation reality.

Code, on the other hand, accompanied by tests, is a good model (an *executable model*) for clearly and unambiguously expressing system behavior in detail. The UML is much better used sparingly to capture only key classes and relationships, and critical execution paths [8].

Experienced OO Guides

If you have not climbed a mountain before, you should really bring a guide who knows the slopes, unless you lust for the thrill of danger. Similarly, a serious OO development effort should really have at least one, if not a few developers with extensive OO experience. If no expert is available, plan to iterate and redesign quite a lot (as discussed above) as your development team learns the ropes, or expect to churn indefinitely during integration and test, trying to patch a fragile, naïve design.

Not surprisingly, teams using OO design for the first time usually fall back on familiar, non-OO patterns. Each developer defines a large controlling class for the developer's area of responsibility that is all function (a *functoid*) and encapsulates no data. The data is thinly wrapped in other classes that contain only the data and access operations (*datoids*). This design is essentially functionally oriented, only superficially structured into classes and objects.

Conclusion

The recommendations in this article are not really new, but their descriptions here contrasted with typical pitfalls seen in government contracting may help you to better understand how to apply them to good effect.

OO design and evolutionary development fit well together. Many of the pitfalls discussed here are characteristic of programs practicing waterfall-style processes. Attaining the potential benefits of OO development is more difficult in a waterfall process. The introduction of OO to an organization should change both its development practices and the designs it produces for systems. If an organization makes only superficial changes (draws more diagrams or uses a different programming language) then what was the point of *changing to* OO development?◆

References

1. Software Productivity Consortium. "Object-Oriented Approach for Software-Intensive Systems (OOASIS)." SPC-2000001-MC. Herndon, VA: SPC, 2000 <www.software.org/membersonly/ooasis>.
2. Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
3. Cockburn, Alistair. Writing Effective Use Cases. Addison-Wesley Professional, 2000.
4. Foote, Brian and Joseph Yoder. "Big Ball of Mud." Fourth Conference on Patterns Languages of Programs, Monticello, IL, Sept. 1997 <www.laputan.org/mud/mud.html>.
5. U.S. Department of Defense. "DoD 5000 Series." Washington: DoD, 2003 <http://akss.dau.mil/darc/darc.html >.
6. Blackburn, Mark, Aaron Nauman, Bob Busser, and Bryan Stensvad. "Defect Identification with Model-Based Test Automation." Herndon, VA: Software Productivity Consortium, 2002 <www.software.org/pub/taf/downloads/SAE_2003.pdf>.
7. Binder, Robert. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, 1999.
8. Ambler, Scott. Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. John Wiley & Sons Canada, Ltd., 2002.

About the Authors

Rich McCabe is a principal member of the technical staff at the Systems and Software Consortium (formerly the Software Productivity Consortium). McCabe co-authored the consortium's Object-Oriented Approach to Software-Intensive Systems (OOASIS) methodology. He also has headed the consortium's pioneering work in the product-line approach for systematic reuse since its inception in the early 1990s. Outside the consortium, he has nearly 15 years of software and system development experience with Bell Laboratories and other firms.

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 20170-4227
Phone: (703) 742-7289
Fax: (703) 742-7200
E-mail: mccabe@systemsandsoftware.org

Michael Polen is a senior member of the technical staff at the Systems and Software Consortium (formerly the Software Productivity Consortium). He co-authored the Consortium's Object-Oriented Approach to Software-Intensive Systems (OOASIS) methodology and consults with consortium members on their practice. Lately, Polen has been merging OOASIS with agile techniques. He has more than 14 years of software and system development experience with Motorola, Booz, Allen and Hamilton, and other firms.

Systems and Software Consortium
2214 Rock Hill RD
Herndon, VA 20170-4227
Phone: (703) 742-7281
Fax: (703) 742-7200
E-mail: polen@systemsandsoftware.org

Identifying Your Organization's Best Practices

David Herron and David Garmus
The David Consulting Group

As organizations strive to improve the design, development, and delivery of software solutions, an effective combination of processes, tools, and methods is critical. Finding the right combination requires careful analysis of the variables that influence gains in productivity and quality. By utilizing industry benchmark data and performance indicators, organizations can collect and analyze their own data to determine the combination of processes, tools, and methods that provide the greatest impact on their performance. This article reports on three client case studies that incorporated various measurement techniques to determine best practices.

Characterizing an organization's best software development practices can optimally be described as those software development practices that yield favorable results, which are often measured by customer satisfaction, reduced time to market, decreased cost and better product quality. This article will discuss how three different organizations used a combination of quantitative measures and qualitative values to identify their best practices. Based on the knowledge gained, the organizations used the results to improve their development practices and/or to advance their process improvement programs.

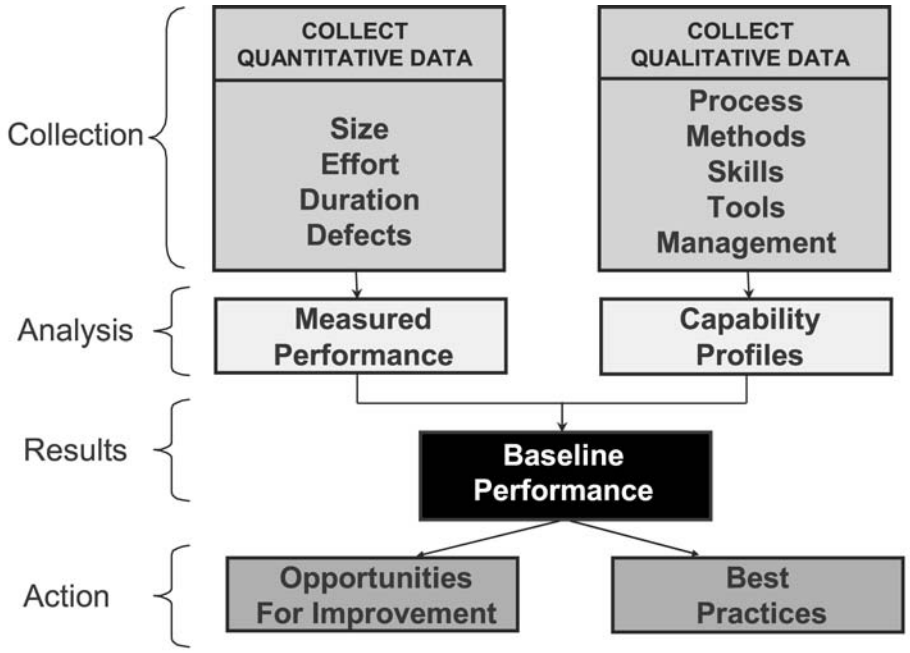
In each case, the desire to identify their best practices was driven by senior level management who wanted results that would have a direct impact on stated business goals and objectives. A summary view of their business goals included the following:

- Reduce project costs (mostly labor).
- Improve their time-to-market delivery of software.
- Minimize defects delivered.
- Improve performance relative to industry benchmark data points.

In all three cases (and in many other companies), the organizational strategy to achieve these goals was centered on quick-fix approaches. Cost reduction frequently tops the list and is usually the driving force behind the decision to outsource software development to an offshore provider. Time to market is often reduced by delivering fewer features to the end user, thus reducing the development work load. Defect minimization is too often ignored. We know too well that quick-fix remedies are not usually effective. However, the alternative to achieving sustained and measurable improvement can be a hard pill to swallow. In order to achieve the findings and the results noted in the cases that follow, senior management had a well-defined vision of what they wanted to accomplish and had to marshal the resources necessary to realize the desired results.

The ability to properly set management expectations and to gain their support was enhanced by the introduction of a measurement model that objectively and quantitatively generated meaningful results.

Figure 1: *Basic Measurement Model*



Introducing the Measurement Model

The key to successful performance management is performance measurement. As the software industry grows out of its infancy into a more mature set of practices, the inclusion of performance measurement to manage and direct decisions is becoming more of a mainstream practice. Organizations long ago recognized the need to establish strategic goals and objectives; equally important, however, is the identification of an appropriate set of measures that provide quantitative evidence that those goals and objectives have been achieved.

A basic measurement model that has been advanced by the Practical Software and Systems Measurement program suggests that an organization follow these three steps:

- Identify the needs of the organization.
- Select measures appropriate to measuring whether the needs have been met.
- Integrate measurement into the software development process.

In each of the three cases examined in this article, the management championing the initiative had identified the needs of their organization. Our effort began when management requested our help in selecting the appropriate measures and creating a measurement model that would result in the quantification of process perform-

ance levels. Furthermore, we were called upon to utilize a measurement model that would provide the ability to compare internal performance measures to industry benchmark levels of performance.

The basic measurement model we used included the collection and analysis of both quantitative and qualitative elements (see Figure 1). The quantitative elements included four basic measures: size, effort, duration, and defects. The qualitative elements included a variety of data points that were used to evaluate levels of competency regarding process, methods, skills, automation, technology, and management practices.

Collected on a project-by-project basis, quantitative data can be displayed in a measured profile that indicates how well a project is performing. Standard industry measures such as function points (FPs) per person month (PM), defect density and time to market must be calculated. If FPs are used to measure project size, there is an opportunity to make comparisons to industry data points that are also based on FPs.

The qualitative data (again collected on a project-by-project basis) results in a matching capability profile. This profile data identifies the attributes that contribute to high or low yields of performance, such as those indicated in Table 1.

These two elements (quantitative and qualitative) come together to form what is commonly viewed as an organization's baseline of performance. The baseline values are compiled from a selection of measured projects and represent the overall performance level of the organization.

Results vary significantly. Some projects perform very well (i.e., they have low cost and high quality), and other projects do not. The quantitative data provides senior management with an objective view of current performance levels. The qualitative data provides the opportunity to examine the attributes of the projects to determine why certain projects have outperformed others. This analysis effort leads an organization to the identification of their best practices and opportunities for improvement.

The following three case studies used this baseline approach in one form or another. The presentation of the results for each of the case studies varies due to the nature of each unique engagement and how the client wanted the information displayed. There is no magic or silver-bullet discovery (as you will see). Basic measures, applied through a practical baseline model, provided senior management with the information they needed to make a more informed decision.

Management	Definition	Design
• Team Dynamics • High Morale • Project Tracking * Project Planning * Automation • Management Skills	• Clearly Stated Requirements • Formal Process • Customer Involvement • Experience Levels • Business Impact	• Formal Process • Rigorous Reviews • Design Reuse • Customer Involvement • Experienced Development Staff • Automation
Build	**Test**	**Environment**
• Code Reviews • Source Code Tracking • Code Reuse • Data Administration • Computer Availability • Experienced Staff • Automation	• Formal Testing Methods • Test Plans • Development Staff Experience • Effective Test Tools • Customer Involvement	• New Technology • Automated Process • Adequate Training • Organizational Dynamics • Certification

Table 1: *Capability Profile Attributes*

Case Study 1: Large Financial Institution

Objective: Identify characteristics of high performing projects.
Scope: Conduct an organization-wide baseline study.
Collection and Analysis: Data (quantitative and qualitative) was collected on 65 completed projects. Productivity rates, expressed in terms of FPs per staff month, were calculated along with three other base measures: duration, resource utilization, and cost. The results were divided into two categories (high performing projects and low performing projects), and an average was calculated for each category as indicated in Table 2:

The data demonstrated that high-performing projects produced (on average) more functionality (148 FPs) in a shorter timeframe (five months) with a modest increase in staffing levels to 2.4.

Qualitative data (attributes about each project) was collected, and profiles of performance were developed that identified characteristics consistently present in the higher performing projects, but limited or absent from the lower performing projects. These sets of attributes were then considered to be the leading factors that contributed to higher performing projects.

Data Collected	High	Low
Average project size in FPs	148	113
Average duration (in calendar months)	5.0	7.0
Average rate of delivery in FPs/PM	22	9
Average number of resources	2.4	1.8

Table 2: *Case Study 1 Results*

The findings listed in Table 3 below indicate the attributes and their frequency of occurrence (percent) in the high- and low-performing projects.

Case Study 2: Midsize Insurance Organization

Objective: Benchmark comparison to industry averages and best practices. Identify best practices opportunities.
Scope: Conduct a baseline study using 25 selected projects.
Collection and Analysis: Measurement baseline data was collected and analyzed to produce performance indicators such as those in the first case study. After deter-

Table 3: *Case Study 1 Findings*

Attributes	High	Low
Development staff very experienced with the design methods used	100%	33%
Full agreement on project deliverables, methodologies, and schedules	100	33
No staff turnover during the project	100	50
Project management experience high	100	50
Formal processes used to gather requirements	100	67
Requirements clearly stated and stable	100	67
Fully automated source code management	100	67
Projects were not impacted by legal or statutory restrictions	67	33
Structured data analysis performed	67	33
Highly experienced analysts and designers	67	33
Significant reuse of code	67	33

Data Collected	Client	Industry Average	Best Practices
FP Size	567	567	567
Productivity (FP/PM)	6.9	7.26	22.68
Duration (months)	12	14	10
Defects/FP	0.24	0.12	0 .02

Notes:
1. FP size represents the average function point size of the delivered product. Some organizations use this as a measure of value (functionality) being delivered to the end user.
2. Duration represents the overall calendar duration of the project from requirements through to customer acceptance.
3. Defect density is a function of defects per function points. A lower number represents fewer defects per function point in the delivered product.
4. Industry data was determined from The David Consulting Group's database of productivity performance measurements for over 7,450 software development projects (from 2001-2003).

Table 4: *Case Study 2 Performance Indicators Comparison*

Common Occurrences	Infrequent or Absent
Historical-based estimating	Formal reviews and inspections
Experienced project managers	Defect tracking
Highly skilled engineers	Effective requirements gathering
Good user involvement	Formal test plans
Formal methodologies	
Rigorous testing	
Well-defined training curriculum	
Effective Software Development Life Cycle	

Table 5: *Case Study 2 Presence or Absence of Attributes*

mining the current level of performance, a comparison to industry average and industry best practices benchmarks was conducted. The results are shown in Table 4.

We examined these data points and analyzed the underlying profile data. Within this sampling of projects, the client's productivity rate was close to the industry average (6.9 versus 7.26); however, plenty of opportunity for improvement still existed as evidenced by the best practices benchmark. The client was actually delivering products (on average) in a shorter timeframe than industry average, and again there was opportunity to improve as the organization moved towards best practices thresholds. Finally, the level of quality (defect density) was significantly below industry data points.

Looking at the findings in this picture (see Table 5), we observed an organization that was getting their software product out the door quickly by increasing staffing levels and shortcutting quality practices. This was further substantiated by evaluating the attributes that were the most common and those that were the most conspicuous by their absence.

These common occurrences in Table 5 refer to practices that the client was already executing. Our analysis suggested that if the client were to focus on the infrequent or absent practices noted in Table 5, they would see a substantial improvement in their level of quality without sacrificing productivity or duration.

Case Study 3: Large Service Organization

Objective: Identify impact of moving to the Software Engineering Institute's Capability Maturity Model® (CMM®) Level 3.
Scope: Perform baseline measures on a sample set of representative projects.
Collection and Analysis: The final case study involved an organization that wanted to estimate the impact that a CMM Level 3 process improvement initiative would have on their performance. They attributed the process areas associated with CMM Level 3 to best practices. To model this improvement, the organization had to first determine its current baseline of performance and establish a composite profile of contributing attributes.

Project data was again collected and analyzed. Averages for size (FPs), productivity (FPs per effort month [EM]), duration (calendar months), and cost (labor) were computed. Using a composite profile, a mapping of the current project attributes for the organization was developed. In parallel, another model was developed for projects of a similar size with a mapping of attributes that matched a CMM Level 3 organization. A modeling tool – Predictor from DDB Software, Inc. – was used to accomplish the modeling. Predictor contains a series of algorithms that are used to calculate productivity levels such as those noted in the findings in Table 6 for the CMM productivity improvements. The values within Predictor are based upon the statistical analysis of software process attributes from more than 8,700 client projects.

Table 6: *Case Study 3 Productivity Levels*

	Baseline Productivity	CMM Productivity Improvements	Impact
Average Project Size	133	133	
Average FP/EM	10.7	24.8	+132%
Average time to market (months)	6.9	3.5	-50%
Average cost/FP	$934.58	$567.29	-40%
Projected Defect Density	0.0301	0.0075	-75%

The projected impact of CMM Level 3 practices for this organization was significant. For the same size project, productivity (FP/EM) was projected to increase by 132 percent, time-to-market reduced by 50 percent, cost reduction by 40 percent and defect density reduced by 75 percent. This modeling technique helped this organization evaluate the potential benefits of CMM process improvement.

The potential impact indicated above may appear to be dramatic, but that is a matter of perspective. Certainly, this significant gain in productivity and reduction in defects would exceed most expectations; however, if the baseline productivity were dramatically below industry averages based on the nature of the process profile, then clearly large gains could and should be expected.

In Summary

These three case studies exhibit a variety of ways in which measurement data can be used to learn more about the following:

- An organization's level of performance.
- Key factors that contribute to high or low productivity yields.
- The level of performance as compared to industry data points.
- The potential impact of strategic initiatives through the use of performance modeling.

Utilizing a measurement model that includes both a quantitative perspective and a qualitative perspective is most important. It is from this vantage point that an organization can access both the measured performance profiles along with an understanding of the process profile elements that contributed to the results. The process profiles have the added advantage of recommending a

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

direction for future improvement strategies.

In reviewing the experiences and results from these three client case studies, readers should not assume that similar outcomes would be achieved in their organizations. The prudent action would be to take your own measures and create your own organizational performance baseline. Utilizing industry-accepted measures such as FPs will allow you to perform the necessary comparative analysis. The investment in a baseline study is relatively insignificant in comparison to the value of the information gained. Of course, the return on that investment can be realized only with the proper execution of improved software development practices.◆

About the Authors

David Herron is co-principal and co-founder of The David Consulting Group. He is an acknowledged authority in using metrics to help organizations monitor the impact of Information Technology (IT), the advancement of IT organizations to higher levels on the Software Engineering Institute Capability Maturity Model®, and the governance of outsourcing arrangements. Herron assists clients in establishing software measurement, process improvement, and quality programs and to enhance their project management techniques. He has more than 25 years experience in managing, developing, and maintaining computer software systems. Herron serves as a Cutter Consortium Expert Consultant and is past chair of the International Function Point Users Group (IFPUG) Management Reporting Committee, a member of the IFPUG IT Performance Committee, and a member of the American Society for Quality. He is a co-author of "Measuring the Software Process: A Practical Guide to Functional Measurement," and "Function Point Analysis: Measurement Practices for Successful Software Projects," and has contributed numerous articles to industry publications and lectured worldwide on functional measures. He attended Union College and Northeastern University.

The David Consulting Group
19 Point View DR
Medford, NJ 08055
Phone: (609) 654-6227
Fax: (609) 654-2338
E-mail: dcgherron@comcast.net

David Garmus is co-principal and co-founder of The David Consulting Group. He is an acknowledged authority in the sizing, measurement, and estimation of software application development and maintenance. He has more than 25 years of experience in managing, developing, and maintaining computer software systems. Concurrently, he served as a university instructor in computer programming, system development, information systems management, data processing, accounting, finance, and banking. Garmus is past president of the International Function Point Users Group (IFPUG) and a member of the Counting Practices Committee. He previously served IFPUG as chair of the Certification Committee, as chair of the New Environments Committee, and on the Board of Directors as director of Applied Programs and vice president. Garmus is co-author of "Measuring the Software Process: A Practical Guide to Functional Measurement," and "Function Point Analysis: Measurement Practices for Successful Software Projects," and has contributed numerous articles to industry publications and lectured worldwide on functional measures. He has a Bachelor of Science from the University of California Los Angeles and a master's degree in business administration from Harvard University.

The David Consulting Group
1935 Salt Myrtle LN
Orange Park, FL 32003
Phone: (904) 269-0211
Fax: (904) 215-0444
E-mail: dcg_dg@comcast.net

Application-Specific Knowledge Bases

Dr. Babak Makkinejad
Electronic Data Systems

It is suggested that companies create and maintain a searchable knowledge base to capture specific issues that are encountered and resolved during commercial off-the-shelf and custom software development projects. Such a knowledge base could benefit the current development staff, the future sustainment staff, and the testing staff, hence facilitating further evolution of the system during its life cycle.

In this article, I will discuss an activity related to both commercial off-the-shelf (COTS) and custom software development that I believe is not often addressed in practice. This activity, often ignored, is the capture, sustainment, and viewing of application-specific knowledge that is generated and acquired by the development associated staff during the course of a development project for the benefit of the sustainment programmers and staff. I will also discuss enablers for addressing this shortcoming.

Undocumented, Poorly Documented, and Work-Around Features

Before proceeding further, I need to define what I mean by COTS and custom software. By COTS software development, I mean customization of COTS systems such as Microsoft Office Suite, Siebel Systems Customer Relationship Management software suite of products, SAP, TeamCenter Enterprise (aka Metaphase) from Unigraphics Solutions, and so on. By custom software development, I mean the development of a software system from scratch but admitting the incorporations of some third-party COTS products such as (class) libraries, application frameworks, and so on. I will note here that, as others have observed, the distinction between COTS and custom is not rigid; a C language compiler and linker may be viewed as the simplest (lowest) COTS application.

Regardless of the rigor in the definition of COTS versus custom applications, it has been my observation that the development staff encounters and resolves a number of issues during the course of the development phase that are of great value to the sustainment phase of the system's life cycle. The development staff often takes advantage of undocumented, poorly documented, or defect work-around features involving the following:

1. COTS Application Programming Interfaces (API).
2. COTS Data Model.
3. Other (third-party) COTS API and Data Models.
4. Vendor-/Version-Specific Operating System.
5. Vendor-/Version-Specific Middleware (class libraries, frameworks, etc.).
6. Network Protocol and Environment.
7. Vendor-/Version-Specific Backend Database.

This is done to meet the business/technical requirements of the software system being constructed. Often, there is no other way than to use these features to meet the delivery deadlines of the system. And, contrary to a popular belief among development managers, COTS application development is also subject to the above considerations.

More crucially, the evolution of the ingredients that have gone into building the system, from software add-ons to the operating environment, may cause some or all of the undocumented features, poorly documented features, or work-around to no longer pertain to a new release of a particular building block of the system. Consequently, the system may undergo significant performance degradations or cease to properly function altogether. The existence of such a knowledge base will go a long way to expedite the resolution of such problems in production.

You must also note that it is most often the case that the knowledge of these features and development *shortcuts* becomes disseminated among the development staff as a sort of tribal knowledge. Unfortunately, this tribal knowledge is usually lost when the system is turned over from the development staff to the sustainment staff and the development tribe is reassigned, or dissolved (which is more likely the case). Thus, further evolution of the system during its sustainment phase could be compromised due to the unavailability of this knowledge base of tribal *know how*.

The usage of these undocumented, poorly documented, or defect work-around features is normally not captured in the technical design documentation. The technical design documents are seldom revised during the course of a typical development project and often are not detailed enough to even provide space for capturing this type of knowledge.

In programming practice, some of this knowledge could be available in the form of source file comments. Even then we are facing the challenges that the individual developer may or may not have provided useful comments, or that the comments may or may not be relevant to the current code revision. Additionally, when these features are in different technical areas, for example API as opposed to data model, they cannot, even in principle, be captured as part of the code (the code-as-documentation crowd notwithstanding). In fact, in such cases, often different individuals or groups are leveraging these undocumented/poorly documented features and thus, are unaware of one another's work. This makes capturing this information in a common technical documentation and format more challenging, but at the same time, more crucial.

Application-Specific Knowledge Bases

It is suggested that an effort be made to develop a knowledge base for the system being built during the development phase of the system. This knowledge base should be able to capture the following:

1. Undocumented features used in developing the system.
2. Poorly documented features used in developing the system.
3. Defect work-around features used in developing the system.
4. Version/patch information of the building blocks of the system.
5. Implemented kludges.
6. Discussion of the technical reasons for using these features.
7. Things that could go wrong if these features cease to work in a future release.
8. Recommendations for replacing these

features in future releases.
9. Date and time stamp for all entries in the knowledge base.
10. Preferred methods for incorporating features used in developing the system.
11. Standards and guidelines that should be used to govern system development.

This type of knowledge base must be distinguished from a frequently asked questions or a technical design document. It should be thought of as an *undocumented corner* magazine column, a pitfalls list, or a compilation of *programmer's shortcuts*. It can be thought of as a small analogue of the large, product knowledge bases that vendors such as IBM, Oracle, Sun, Microsoft and others supply.

While such large knowledge bases are for commercial products, the knowledge bases discussed in this article pertain to specific applications produced for specific clients and hence the phrase: *application-specific knowledge bases*. And, in an analogous manner, these application-specific knowledge bases are conceived to be easily accessible and searchable.

An application-specific knowledge base is thus an enabler for expediting the resolution of defects and in assessing the risks involved in the evolution of specific systems as requirements, building blocks, and operating environment evolve over time. Both the technical development staff and the business staff will be consumers of this information for technical and business purposes.

This type of knowledge base must be distinguished from a defect tracking/help desk system. Although some or all of the issues and knowledge captured in application-specific knowledge bases may already exist in a defect tracking system, they are not there in a usable format. More information on this follows.

Challenges to Adoption

There are two challenges with realizing the above vision. Perhaps the most significant barrier to adoption of this practice as part of the software engineering process is the development staff buy-in. In many cases, members of the development staff are required to supply/update documentation that, in their minds, has no value. In fact, many development staff members consider much of the documentation effort misplaced and non-value-added.

Another major impediment to developer buy-in is the time factor; development staff is typically under so much time pressure that they cannot find time to create what they perceive as an additional system (the knowledge base) beyond the system they are funded and allocated to create. Additionally, in many instances the development staff is not going to be around for the sustainment phase. It is a leadership challenge to motivate the development staff and inspire them to do a great job and to facilitate the activities of the sustainment staff.

Perhaps the best way to proceed is to periodically ask the development staff to supply a list of items that they feel will be important to know for the sustainment phase in free form. Then as part of the process of software documentation, one can compile these inputs into a knowledge base as part of the key deliverables of the system.

The other challenge is the packaging and delivery of this knowledge base. For the knowledge base to be useful, it must satisfy the following criteria:

"The technical design documents are seldom revised during the course of a typical development project and often are not detailed enough to even provide space for capturing this type [undocumented, poorly documented, or work-around] of knowledge."

1. **Be easily accessible.** The user should be able to get to the knowledge base without having to navigate a hierarchy of network folders or Web-based pages. Nor should he or she have to look for a specific file among numerous electronic documents for the information that he needs.
2. **Be easily searchable.** The user should not have to use Global Regular Expression Parser or the Search feature of the operating system to look up the information for which he is searching.
3. **Be easily navigable.** The user should be able to easily and painlessly move from one item to the next relevant item.
4. **Be easily updateable.** The knowledge base should be easily updateable to reflect the changes to the software, its building blocks, and its operating environment.
5. **Be portable.** The user should be able to view the information without having to be hooked to the enterprise network; it must be accessible in a disconnected mode from a portable computing device. This is an essential feature for certain class of applications that require on-site support.
6. **Be secure.** Unauthorized access to use the knowledge base for malicious purposes must be prevented.

One approach will be to utilize the same COTS tools that have been used for building, updating, and maintaining the software's help system to deliver such an application-specific knowledge base. These tools are often multi-platform, thus enabling the development of a knowledge base that can be made to satisfy all of the criteria above. These types of systems are easily searchable, often use hypertext to provide navigable links, can be edited by using a word processor, and are portable. Unfortunately, this is a heavyweight approach since it requires knowledge of the specific help system creation tool, and the effort itself will become part of the cost of the development.

The next best candidate for the deployment of such a knowledge base will be to leverage an existing defect tracking/help desk system by augmenting it with the development staff's issues and resolutions. With this approach you have the added challenge of customizing the defect tracking system to distinguish among generic defects and the application-specific *gotchas* of the knowledge base. It must keep track of the requirements, building block revisions, patches, and changes to the operating environment that pertain to the items captured in the knowledge base.

Unfortunately, even though most tools are, in principle, capable of satisfying a number of the above criteria, they do not satisfy them all. Specifically, in the areas of navigability and portability, they leave much to be desired. While the navigability criterion may be addressed with the addition of *Google-like* features, the portability will always be an issue for most of these centralized systems.

The simplest approach, which is quite doable and lightweight to develop and deploy, is to create a hypertext markup language (HTML) document that will contain the knowledge base. In this approach, all that is needed is a text editor and staff who are knowledgeable in HTML; expensive development tools will not be required. This approach satisfies the first

five criteria above.

The three approaches above all require additional developer interactions that are specific to the knowledge base. They add to the development effort and cost. There is an alternative approach to developing the knowledge base that leverages a common activity that occurs during the course of a development project; that is, team members send one another e-mails discussing the problems they need to solve, how to work around limitations of the tools, etc. In fact, on the Internet, the archival and current material in technical discussion forums serves exactly the same purpose. I am suggesting archiving the e-mail exchanges of developers for the sustainment phase.

This developer e-mail base, plus the other documents that are created and assembled by the development team is, in effect, a knowledge base that can be a resource for sustainment staff. To leverage this naturally created knowledge base, you need a generic, straightforward method to capture this source of information at the end of a development project and make it available to and easily searchable by the sustainment team.

This would entail using the built-in archiving functions of the messaging sever at project start-up, followed by the search capabilities of the e-mail client itself such as Microsoft Outlook on a *.pst* file. In fact, there are currently e-mail clients that support something akin to Google to search e-mails, for example, Bloomba at <www.statalabs.com>. In this manner, you may enable the sustainment staff to quickly find information (or, at least, clues) about how specific problems were solved by the development staff.

The sixth criterion, the security requirement, is a challenge for all approaches. For those applications that will have their own security features, access to the knowledge base may be controlled by leveraging the application's own security features. In other cases, you must consider the details of secure access to the knowledge base and assess the risks involved in unauthorized access to this data. While 100 percent security is not even theoretically achievable, a judicious approach to access control and distribution lists should largely mitigate the security concerns related to the knowledge base.

Conclusion

I believe it a good idea to canvas the development staff for issues that pertain to using undocumented, poorly documented, defect work-around, kludges, and gotchas that have been encountered and/or leveraged during the course of system construction for both COTS and custom application. I further conceive of the utility of compiling that input into an application-specific knowledge base for consumption by the technical and business staff during the sustainment phase of the system.

Although there are multiple cost-effective techniques for packaging and enabling such a knowledge base based on available commercial tools, I favor capturing e-mail exchanges as the most lightweight method. Since the amount of data in the knowledge base is necessarily limited to the development phase, *key* word searches will not be as tedious as on the Internet since the number of hits would be either small or none at all.

Finally, I believe that human factor issues are the greatest barrier to developing and deploying such a system. I respectfully urge the project leadership in the information technology industry to make a concerted effort to supply the necessary positive motivations for this effort to become practicable. We owe this to those who come after us to sustain the systems that we are producing today.◆

About the Author

Babak Makkinejad, Ph.D., is a consultant with Electronic Data Systems. He has worked in the areas of computational physics, computer graphics, image processing, and enterprise software development. Makkinejad has a doctorate in theoretical physics from the University of Michigan in Ann Arbor.

Electronic Data Systems
5555 New King ST
Troy, MI 48098
Phone: (248) 696-2311
Fax: (248) 696-2590
E-mail: babak.makkinejad@eds.com

More Online from CrossTalk

CrossTalk is pleased to bring you additional articles with full text at <www.hill.af.mil/crosstalk/2005/06/index.html>.

Knowledge Management and Process Improvement: A Union of Two Disciplines

Gregory D. Burke
Federal Aviation Administration
William H. Howard
Northrop Grumman Mission Systems

The experience at the Federal Aviation Administration (FAA) shows that process improvement and knowledge management complement each other well. Process improvement helps the organization increase its effectiveness through continuous examination with a view to doing things better. Once processes are documented, roles and responsibilities are readily identified and associated activities are performed. Legacy processes are modified to reflect organizational changes. Knowledge management facilitates communication among organizations, increasing information sharing and utilizing process documentation. This information sharing promotes organizational unity and allows FAA headquarters and regional operations to function efficiently.

Connecting Earned Value to the Schedule

Walt Lipke
Tinker Air Force Base

For project cost, analysts can predict the final value with some confidence using the Independent Estimate at Completion (IEAC) formulas from Earned Value Management (EVM). However, EVM does not provide IEAC-like formulas by which to predict the final duration of a project; many express the opinion that schedule information derived from EVM is of little value. This article discusses the problem and develops a methodology for calculating the predicted project duration using EVM data. The methodology uses the concept of Earned Schedule and introduces an additional measure required for the calculation.

Process Therapy

Paul Kimmerly
Defense Finance and Accounting Service

Over a year ago, the author suffered a spinal cord injury that required surgery, an extended hospital stay, therapy, and a trip into the medical sub-culture. This article looks at process improvement and relates it to the author's experiences recovering from his injury. If an organization's process is injured, process therapists using the diagnostic and treatment tools at their disposal can help lead an organization to recovery.

In June 2003, I suffered an unusual spinal cord injury. A blood vessel burst inside the cord on the first day of a three-week vacation in Alaska. I ended up in the hospital in Anchorage and then was flown home to Kansas City. The author promises this will not deteriorate into a maudlin journey of self-discovery, but if it does, readers are encouraged to send angry messages to the author.

The initial diagnosis was pretty grim. It looked like I would be paralyzed from the waist down, but after flying home and having surgery to remove the resulting blood clot from the cord, the diagnosis changed. I started to hear things about taking small steps, measuring progress and focusing on continuous improvement with no certainty of the end result. It sounded very familiar.

At work, I have served as a member of the Software Engineering Process Group (SEPG) for the last 10 years. Suddenly, I had to apply all the tools from my role as an SEPG member to myself. Talk about a rude awakening! Before this, I could tell other people what was wrong and leave them to fix it. Now, I actually had to do all the things I told other people to do.

I found that the therapy process involved in recovering from physical injury maps closely to the process involved in healing an organization's process injury. A therapist's tools of empathy, encouragement, humor, challenge and success in the achievement of others fit a process improvement change agent's role perfectly.

This article will show how planning, a good therapist, collaborative effort and measurement all play a role in helping organizations recover from a process injury.

Background

Before my injury, I ran, played softball, and played soccer. My wife and I spent our annual vacations camping and hiking in national parks. Suddenly everything changed. I received a sudden shock that a system or a process could break down unexpectedly. Organizations that have successfully put out software can experience the same thing when they see results slipping, experience a major system crash, suddenly miss a major deadline or lose a customer. That's when the questions start: What happened? Wasn't everything okay? How could that happen? Why didn't I see any signs?

Diagnosis

The first thing to do in the case of an injury is to diagnose what happened. In my case, that involved MRIs, CT scans, and surgery. For an organization, a process assessment and a process improvement model are needed. In the case of a physical injury, doctors compare diagnostic results with the models they have of how the human body should work. The Capability Maturity Model® (CMM®) for Software and CMM Integration℠ provide models of how an organization should work. The diagnostic tools associated with each of these models are the assessment methods. At first, expert diagnostic help is important. However, an organization should not rely entirely on outside help. It should train some of its own people in the model and provide them experience in using the assessment methodology.

Assessments can be formal or informal. In the initial stages of diagnosis, an informal assessment can provide a good first look at identifying problem areas. Rather than focus on the process improvement models, an organization should focus on its problems. By addressing the problems, satisfaction of the selected model comes more naturally. As I started to get some function back in my legs, I did not need to know that I had inadequate ankle dorsiflexion. I needed to know the exercises to strengthen the muscles that bend my ankle. The models are best used as references to help address problems.

Treatment/Therapy

Immediately following my surgery, I began a long road of therapy to literally get back on my feet. It was obvious from the beginning that my doctors and therapists had a plan for my recovery. Once again, it all sounded very familiar to me. In fact, one day my boss was visiting me when my therapy doctor came through. The doctor explained that they established a scoring chart for therapy patients to evaluate when they were ready to go home. He also stated that he and I would meet weekly with the head nurse, my insurance caseworker, and my occupational and physical therapists. The meeting would review every aspect of my case and update my score against their plan for my recovery.

My boss said, "Don't get him started." After all, our SEPG had been the one to encourage meeting management and using metric information to make decisions. I think he was afraid I might try to improve the hospital's processes.

The need for process therapy often stems from a trauma of some kind. An organization's immediate reaction is that the situation needs to be fixed. This might create a flurry of activity that is short-lived if a quick fix is found. The key for the process therapist is to find the underlying reasons for the trauma and address them. By doing this, the therapist can build a foundation for long lasting improvement.

My therapy began immediately with an evaluation by both therapists. Each of them measured what I was physically capable of doing before we got started. This gave them a baseline of what to expect and provided a mark against which they could compare my improvement. It bore great similarities to taking an initial snapshot of where a project stands, as determined by the *diagnostic* assessment, before embarking on rehabilitating its processes. The therapists had a target goal in mind, just as an organization often has a target maturity level or performance goals. As change agents, the therapists laid out a series of steps to get me on the right track.

Process therapists must also focus on the steps an organization needs to build lasting improvement and reach improvement goals. Whether those goals relate to a maturity level or a performance goal, a planned approach to addressing specific items is critical. Sometimes these items are not directly related to the process improvement model. Often they involve related organizational issues that affect how the change will happen.

In my case, learning to get dressed was the issue, but I needed to improve my flexibility before I could do certain things. Just

℠ CMM Integration is a service mark of Carnegie Mellon University.

like an organization, I resisted. It took some patience and some prodding on my therapist's part, but I got to where I could stretch as needed.

Process therapists need to use the same kind of sometimes gentle, sometimes firm persuasion to keep an organization moving towards reaching its goal. They also need to look for the underlying reasons for the resistance. The unwritten rules or norms for the organization can come into conflict with the process improvement goals. The process therapist needs to be prepared to deal with any covert resistance.

Open resistance is easier to identify and to address. Covert resistance can sabotage improvement efforts. Unexpected results often highlight covert resistance. Process therapists should monitor results and actions after process improvements are agreed upon and implemented.

A key item that had to be in place was my desire to get better. For process improvement to work in an organization, the organization has to want to change. In my recovery, I had to mentally accept that I needed to change, and I had to focus on getting better. It is the same for an organization. The organization needs a manager who understands the need and sees the benefits of change. By identifying such a champion, the organization will ensure itself of future improvement. The champion can help the therapist overcome resistance by enforcing agreements and voicing commitment to the improvements.

A therapist always likes a willing patient and can use that patient to show others the way to succeed. Unfortunately, not everyone wants to get better. Some think the status quo is fine or the therapy is too hard. That kind of thinking leads to resistance. Resistance comes from a variety of sources and the therapist has to be ready to respond. There were times I resisted my therapists, but they were few. That helped me succeed in my recovery.

I saw a number of patients fighting against their therapists or not doing the work between sessions that the therapists encouraged. As a result, their recovery did not progress as far as quickly. This was evidenced by false starts and repeating the same exercises over and over again. Organizations can see the same kind of results. False starts plague improvement efforts that do not have the necessary champion or commitment to change. The therapist needs to set up an improvement regimen and stick to it. Even if the organization wants to resist, the therapist must continue to exhibit the desired behavior and stick to the plan.

A little challenge can be a healthy thing for the patient. Process therapists need to challenge the organization to keep going one step further to ensure continuous improvement. A challenge can also help push an organization through plateaus. My physical therapist was sometimes subtle; when we would walk, he would stay just ahead of me to make me pick up my pace and try harder to keep up. I knew what he was doing, but it worked anyway.

Process therapists need to find ways to keep the organization moving forward. Regular appointments with managers provide important opportunities to discuss treatment and review progress. It is vital that the process therapists talk to people within the organization and look at measurement data to gather information to present to management. It helps to identify the key issues for a manager and use them to find a way to keep the organization focused on improvement. If a process therapist can find ways to address those issues or provide information about them, a manager will pay close attention.

Another important thing for the patient and the organization is to check their pride at the door. During my hospital stay, it seemed like everybody got to see or know about everything. I had no secrets by the time I left. One nurse tried to make small talk while we worked our way through the situation. It turned out that I worked with her husband. Great! All I could say in that awkward and uncomfortable moment was, "Oh ... uh ... tell Steve I said 'Hi'."

Organizations will have those embarrassing moments too. An organization will find things out that it did not want to know. The organization may find them out in front of a senior manager. At those times, the process therapist must be supportive and help the organization work through the situation. The difficult times can lead to frustration when a patient will want to shut down. The therapist has to anticipate these situations and plan how to address them. It may come down to helping an organization gather data so it is better prepared when facing a potentially embarrassing situation. If the process therapists can help an organization find out some fundamental information about itself, they can help it use that information to its advantage.

Lastly, the process therapists should be willing and able to demonstrate the desired behavior to the patient to speak from experience and show empathy for what the patient is facing. A good place to start is meeting management. Many organizations suffer from free-form stream of consciousness meetings that lead nowhere. Process therapists should encourage good meeting behavior by using agendas and meeting minutes for any meeting they lead. By demonstrating the desired behavior, a process therapist can influence the organization.

Summary

After my injury, I recognized that I was going through all of the stages I learned to look for in an organization when trying to bring change. Recognizing the similarities between my life-altering event and my work helped me get some ideas for improving myself based on what I had learned bringing improvement to my organization.

Process therapists should look for ways to help an organization improve, just like the physical therapists helped me. The recovery starts with the diagnosis of the problem and a plan to address it. From there, the therapist has to know their patient and create ways to effectively target the small steps in improvement. Throw the model away for a while because the technical terms can scare off a patient. Use terms the patient knows. Talk about problems, not the model's jargon. The achievement of process improvement goals requires buy-in by the organization and persistence by the therapist. If the organization resists or suffers some embarrassing setbacks, the therapist needs to find creative ways to work through them. Above all, a therapist must demonstrate they know the problems and can empathize with the patient.◆

About the Author

Paul Kimmerly has 16 years experience in software development for the different incarnations of the Defense Finance and Accounting Service Technology Services Organization. A member of the Software Engineering Process Group since 1993, he currently serves as the group's chair. Kimmerly is an authorized Capability Maturity Model® Integration℠ Assessment Method for Process Improvement Lead Appraiser. He has written several articles on process improvement for CROSSTALK.

DFAS-KC/TKZ
1500 E 95th ST
Kansas City, MO 64197
Phone: (816) 926-5364
DSN: 465-5364
Fax: (816) 926-6969
DSN Fax: 465-6969
E-mail: paul.j.kimmerly@dfas.mil

You Want Reality Computing? You Can't *Handle* Reality Computing!

It's April. Time for the annual Systems and Software Technology Conference, and I'm sitting on another airline flight writing a BACKTALK column. Some things don't change much. On the other hand, some do.

I am flying to Salt Lake City from Detroit, where I teach college part-time. For the past 10 years, I have flown into and out of Detroit, usually on Delta airlines. For the last nine years, Delta has used the L.C. Smith Terminal – an older terminal, but convenient. To fly out, rental car busses dropped you off near the ticket counter; it was a short walk to the gate. Flying in was a breeze – the luggage pickup was within 50 yards of the rental car pickup.

Last month, in an effort to upgrade, the Detroit airport has expanded its new terminal, and Delta has relocated there. The new terminal is both modern and pretty, but not as functional. You have to walk about a mile to get from the ticket counter to the gate, or back from the gate to the luggage pickup. To get from rental car drop-off to the ticket counter involves walking several hundred yards, negotiating revolving doors while carrying luggage, and two escalators. Once you have your ticket, getting to the gate involves three escalators, four moving walkways, and an underground corridor that has varying mood lighting and sound, supposedly to invoke images of a thunderstorm. Just what I want – dimming lights when I am trying to replace items in my pockets from security, and see how to step on and off of moving walkways.

Modern is not always better. Perhaps I am becoming set in my ways, but new is not always better. As another example, what happened to good old-fashioned humor on television? When I turn on the television, I want entertainment. I miss Seinfeld. The other night on TV, I had my choice of several reality dramas. I haven't figured out the reality of these shows. Personally, I have never been (a) stranded on a deserted island trying to survive without fire, or (b) traversing the African continent without enough frequent-flyer points to get a plane ticket home.

Somehow, knowing there is a film crew and a sound team supporting me would take the actual survival drama out of the situation. I can't figure out why they don't just break one of the cameras and use the metal pieces (or even the batteries) to create sparks and discover fire!

Unfortunately, Seinfeld probably won't return. I'd settle for a good rip-off. It seems, however, that the trend is more and more absurd reality television. Perhaps what we need are topics that are more realistic. How about a reality show that shows the real world? I have an idea for a show about reality computing: It would have scenarios that include the following computing classics, guaranteed to provide huge ratings.

First, how about being able to vote off end-users who can't seem to agree on requirements? Once a week, we get the developers together, and they get to write down the names of the most unsupportive and unresponsive subject matter expert (SME). The SME voted off would have to work for six months testing the effectiveness of varying brands of odor eaters in Iraq.

Next, by the same token, end-users get to vote off the developer who added the most useless feature to their system. Those voted off would be forced to work converting legacy Fortran.

OK, you're right. These ideas are too realistic. Reality shows require some believability, but not too much. What we need are ideas that would provide some reality, but are enough removed from actual life to entertain and amuse. I have a couple of ideas that I think would be funny, but non-realistic enough to not only entertain and amuse, but also draw high ratings. In my first idea, developers are mandated to develop systems using a new language – but no (or few) compilers, tool sets, or trained personnel would be available. There would be incentives to develop supporting toolsets and compilers. Just about the time supporting software and personnel become available, tell the developers, "Just kidding," and entice them not to use the new language. In fact, let them develop mission-critical software with languages that perform no range checking, parameter checking, type checking, or memory protection. Imagine the hilarity and laughs from this hypothetical scenario! There would be a million laughs a minute as developers scramble to track down invalid pointers. The thrill of locating that last uninitialized variable while delivery deadlines slip and costs rise would keep viewers glued to their seats.

For my second idea, let's take obsolete standards for developing software and update them so they actually work. Make them a new standard. Then just as the new standard becomes useful, remove it as a standard and tell developers to simply use best practices. Don't define what the best practices are. Imagine the chaos and the chortles that will result! In fact, to really facilitate the humor, we could cut funding to organizations that provide software support and quality improvement.

In my third idea, top-level managers are tantalized with new processes that aren't processes. Let's give these almost-processes appealing names like "Flexible Methodology" or "Maximum Programming." Show policy makers that these almost-processes work (and work well) for smaller, non-critical applications. Then, tempt managers to apply these almost-processes to large-scale mission-critical programs. It is always a real side-splitting laugh to watch folks learn over and over that, on large-scale mission-critical systems, you can't skip things like formal interface design, configuration management, documentation management, quality assurance, and requirements engineering. The hysterical guffaws from applying lightweight processes and free-fall coding practices on complex systems guarantees a laugh riot every minute!

I don't know about you, but I think the above scenarios – which, granted, are quite far-fetched and unrealistic – would provide just about the humor television needs. In fact, after 31 years of developing, managing, and supporting Department of Defense software, these ideas make me laugh all the time.

Oh yeah. We need a name for our reality show. Funny, the only name I can come up with is "Survivor" – but it's already taken.

– David A. Cook, Ph.D.
The AEgis Technologies Group
dcook@aegistg.com

CrossTalk / MASE
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737